Source Code Interaction on Touchscreens

Inaugural-Dissertation zur Erlangung der Doktorwürde
der Fakultät für Sprach-, Literatur- und Kulturwissenschaften
der Universität Regensburg

vorgelegt von
Felix Raab
München
2014

Regensburg 2014

Erstgutachter	Prof. Dr. phil. Christian Wolff
	Universität Regensburg
Zweitgutachter	Prof. Dr. rer. nat. Florian Echtler
	Bauhaus-Universität Weimar

Dedicated to my family.
Dedicated to Doris, who has joined me on this journey.

Don't generalize; generalizations are generally wrong.

(Butler W. Lampson)

Abstract

Direct interaction with touchscreens has become a primary way of using a device. This work seeks to devise interaction methods for editing textual source code on touch-enabled devices. With the advent of the “Post-PC Era”, touch-centric interaction has received considerable attention in both research and development. However, various limitations have impeded widespread adoption of programming environments on modern platforms. Previous attempts have mainly been successful by simplifying or constraining conventional programming but have only insufficiently supported source code written in mainstream programming languages. This work includes the design, development, and evaluation of techniques for editing, selecting, and creating source code on touchscreens. The results contribute to text editing and entry methods by taking the syntax and structure of programming languages into account while exploiting the advantages of gesture-driven control. Furthermore, this work presents the design and software architecture of a mobile development environment incorporating touch-enabled modules for typical software development tasks.

Zusammenfassung

Die direkte Interaktion auf Touchscreens hat sich zu einer wesentlichen Form der Bedienung von Geräten entwickelt. Die vorliegende Dissertation beschäftigt sich mit der Entwicklung von Interaktionsmethoden zur Bearbeitung von textbasiertem Quellcode auf Geräten mit Touchscreen. Seit der "Post-PC-Ära" spielt Touchscreenbedienung eine wachsende Rolle in Forschung und Entwicklung. Diverse Limitierungen erschweren jedoch die Ausführung und Bedienung von Programmierumgebungen auf modernen Plattformen. Bisherige Arbeiten erzielen vor allem durch die Vereinfachung oder Einschränkung konventioneller Programmierung Erfolge, unterstützen Quellcode von Mainstream-Programmiersprachen allerdings nur unzureichend. Diese Arbeit umfasst die Konzeption, Entwicklung und Auswertung von Methoden zur Bearbeitung, Auswahl und Erzeugung von Quellcode auf Touchscreens. Die Ergebnisse ergänzen Texteingabe- und Bearbeitungsmethoden dahingehend, dass die Syntax und Struktur von Quellcode berücksichtigt wird und gleichzeitig die Vorteile gesten-gesteuerter Bedienung ausgenutzt werden. Darüber hinaus stellt die Arbeit die Konzeption und Software-Architektur einer mobilen Entwicklungsumgebung mit Touch-Bedienung vor.

Acknowledgements

First, I would like to thank Prof. Dr. Christian Wolff for his supervision of my work and for his professional advice. I am particularly thankful that he approved my topic since I could not imagine having completed the present work without an intrinsic interest in this research area. Also, I am thankful that he allowed me to teach University courses covering the areas of programming and user interaction.

Second, I would like to thank Prof. Dr. Florian Echtler for his supervision. I am pleased about having met him during his time as Visiting Professor at the University. Thanks for your comments and support with publications.

Third, I would like to thank Dr. Markus Heckner for having enabled me to conduct research in his Android programming courses. Thanks to Markus Fuchs for his help in carrying out these studies and for various discussions during the many long train journeys to Regensburg. Also, thanks to all former University colleagues for having contributed to a pleasant work environment.

Finally, I would like to thank all participants that took part in my user studies. Your feedback has been valuable, and the results generated by you interacting with my prototypes are a central part of this work.

Contents

Abstract vii

Zusammenfassung ix

Acknowledgements xi

List of Figures xxii

List of Tables xxiii

Note on Writing Style xxv

1 Introduction 1

1.1 Problem Statement 1

1.1.1 Challenges 2

1.1.2 Prior Approaches 2

1.1.3 Research Objectives 4

1.2 Research Approach 5

1.3 Publications 6

1.4 Structure 7

1.5 HCI Terms 9

I Background and Prior Work 11

2 Background 13

2.1 Source Code, Programming, and Usability 13

2.1.1 Programming Paradigms and Languages 14

2.1.2 Cognitive Aspects of Programming 16

2.1.3 Forms of Representation 17
2.2 Integrated Development Environments 21
2.2.1 IDE Components 21
2.2.2 Usability and Usefulness of IDEs 23
2.2.3 Text Editors and IDEs 23
2.3 Types of IDEs 24
2.3.1 The first IDE 25
2.3.2 Textual Environments 25
2.3.3 Modern Desktop Environments 26
2.3.4 Visual Programming Environments 28
2.3.5 Hybrid Environments 29
2.3.6 Recent Developments 31
2.4 Programming on Touchscreens 32

3 Related Work **35**
3.1 Touch and Pen Development Environments 36
3.2 Text Editing 40
3.2.1 Text Editing in Desktop Environments 41
3.2.2 Text Entry on Touchscreens 42
3.2.3 Text Editing Gestures 43
3.3 Editor User Interfaces and Interaction 45
3.3.1 Novel Editor Interfaces 45
3.3.2 Intelligent Code Editing 46
3.3.3 Code Navigation and Search 49
3.4 Commands, Menus, and Gestures 51
3.5 Multi-modal Development Tools 55

II Source Code Interaction **59**

4 Editing Source Code **61**
4.1 Introduction 61
4.1.1 Code Editing Operations 62
4.1.2 Code Editing Triggers 64
4.2 Refactoring 66
4.2.1 Refactoring Tools 69
4.2.2 Gesture-driven Tools 72

4.3 User Study . . . 73
4.3.1 Editor Operations . . . 74
4.3.2 Participants . . . 76
4.3.3 Test Setup . . . 76
4.3.4 Procedure . . . 79
4.3.5 Results . . . 80
4.3.6 Observations . . . 86
4.3.7 Discussion . . . 87
4.3.8 Design Recommendations . . . 89
4.3.9 Conclusion . . . 91

5 Selecting Source Code **93**
5.1 Introduction . . . 93
5.1.1 Terminology and Selection Mechanics . . . 94
5.1.2 Modeless vs. Modal Selection . . . 95
5.1.3 Selection in Desktop IDEs . . . 95
5.1.4 Selection on Mobile Platforms . . . 97
5.2 User Study . . . 100
5.2.1 Participants . . . 100
5.2.2 Test Setup . . . 100
5.2.3 Procedure . . . 101
5.2.4 Analysis . . . 103
5.2.5 Results . . . 106
5.2.6 Discussion . . . 111
5.3 Interaction Methods . . . 112
5.3.1 Syntax-aware Selection . . . 112
5.3.2 Selection Gestures and Widgets . . . 117
5.4 Conclusion . . . 120

6 Creating Source Code **123**
6.1 Challenges . . . 123
6.1.1 Fat Fingers . . . 124
6.1.2 Touch Model . . . 124
6.1.3 Language Model . . . 125
6.1.4 Text vs. Source Code . . . 125
6.2 Code Creation in Desktop IDEs . . . 126

6.2.1 Smart Typing . . . 127
6.2.2 Code Completion . . . 127
6.2.3 Code Hints . . . 128
6.2.4 Code Templates . . . 129
6.2.5 Code Generation . . . 130
6.3 Towards a Code Entry Keyboard . . . 130
6.3.1 General Design Approach . . . 130
6.3.2 Keyboard Layout and Size . . . 132
6.3.3 Gestures and Marking Buttons . . . 136
6.3.4 Code Templates . . . 137
6.3.5 Code Completion . . . 139
6.3.6 Underlying Models . . . 140
6.4 User Study . . . 141
6.4.1 Participants . . . 141
6.4.2 Test Setup . . . 142
6.4.3 Tasks and Procedure . . . 142
6.4.4 Results . . . 144
6.4.5 Discussion . . . 153
6.5 Improvements and Simulations . . . 156
6.5.1 Key Layout . . . 156
6.5.2 Touch Model . . . 158
6.5.3 Language Model . . . 160
6.5.4 Widgets . . . 167
6.5.5 A Revised Model . . . 167
6.6 Conclusion . . . 168

III Design and Implementation 171

7 A Touch-enabled IDE 173
7.1 Device Class and Platform . . . 173
7.2 Approach to Interaction Design . . . 174
7.2.1 Integration of Gestures . . . 175
7.2.2 Conflict Resolution . . . 176
7.2.3 Widget-based Techniques and Menus . . . 178
7.2.4 General Guidelines . . . 180
7.3 IDE Components . . . 180

7.3.1 File Browsing . . . 180
7.3.2 Working Sets, File Sets, and Layouts . . . 181
7.3.3 Navigation . . . 186
7.3.4 Code Entry and Editing . . . 189
7.3.5 Error Highlighting and Code Review . . . 193
7.4 Conclusion . . . 196

8 Software Architecture 199
8.1 Introduction . . . 199
8.1.1 Language Support . . . 200
8.1.2 Presentation . . . 201
8.1.3 Code Analysis . . . 202
8.1.4 Other Modules and Summary . . . 205
8.2 Reference Architectures and Existing Tools . . . 205
8.2.1 The Eclipse Project . . . 205
8.2.2 Syntactic Analysis and Editor Components . . . 209
8.2.3 Semantic and Static Analysis . . . 211
8.3 Concrete Architecture . . . 212
8.3.1 Target Platform and Environment . . . 212
8.3.2 Modules and Events . . . 212
8.3.3 Services and Core Objects . . . 215
8.3.4 Adapters and Bridges . . . 217
8.3.5 Model-View-Controller and Commands . . . 218
8.3.6 UI Components and Gestures . . . 219
8.3.7 Concurrency and Code Analysis . . . 221
8.3.8 Discussion . . . 222
8.4 Conclusion . . . 223

IV Conclusions 225

9 Conclusions 227
9.1 Summary and Contributions . . . 227
9.1.1 Motivation . . . 227
9.1.2 Part I: Background and Prior Work . . . 228
9.1.3 Part II: Source Code Interaction . . . 228
9.1.4 Part III: Design and Implementation . . . 229

9.2 Future Directions 230
9.2.1 Opportunities for Further Work 230
9.2.2 Related Projects 232
9.3 The Future of Programming Environments 234

Bibliography **237**

Appendices **257**

Appendix A Study on Editing Source Code **259**

Appendix B Study on Selecting Source Code **265**

Appendix C Study on Creating Source Code **275**

List of Figures

2.1 Languages, paradigms, environments, and representations 20
2.2 Emacs and Vim 26
2.3 The Smalltalk-80 IDE and Eclipse 27
2.4 GRaIL and Scratch 28
2.5 Barista and IPython 30
2.6 Scope and approach of this work: Touchable source code 33

3.1 Linear, radial, and grid-based menus 53
3.2 A gaze-enhanced IDE: EyeDE 57

4.1 Types of code editing operations 63
4.2 Dimensions of code editing operations 64
4.3 Examples of refactoring menus 66
4.4 Refactoring and AST transformation 68
4.5 Eclipse refactoring errors for *Change Method Signature* 70
4.6 Eclipse refactoring errors for *Extract Constant* 70
4.7 *RefactorPad*: The experimenter's system 77
4.8 *RefactorPad*: The participant's system 78
4.9 *RefactorPad*: Overview of the test setup 78
4.10 *RefactorPad*: Post-task questionnaire 81
4.11 *RefactorPad*: Visualized agreement of gestures 82
4.12 *RefactorPad*: Visualized agreement for *Undo/Redo* 83
4.13 *RefactorPad*: Gesture agreement scores 84
4.14 *RefactorPad*: Goodness–SMEQ–Agreement 85
4.15 *RefactorPad*: Goodness–SMEQ–Agreement 86
4.16 *RefactorPad*: Final gesture set 90

5.1 Selection range, anchor, and head . . . 94
5.2 Text selection on iOS . . . 98
5.3 Text selection on Android . . . 99
5.4 Custom application for replaying selection events . . . 104
5.5 Overview of the custom system for analyzing selections . . . 105
5.6 Relative frequencies of selected AST node types . . . 108
5.7 Selections matching at AST node boundaries . . . 109
5.8 Average number of selected lines for AST node types . . . 110
5.9 Initiating a syntax-aware selection . . . 113
5.10 Touch locations and AST selection boundaries . . . 114
5.11 Node visiting and skipping . . . 115
5.12 Changing the selection range . . . 116
5.13 *Selection spans* . . . 118
5.14 *Selection panning* . . . 119
5.15 *Selection rails* . . . 120

6.1 Challenges of touch-based code creation . . . 126
6.2 Smart typing . . . 127
6.3 Code completion . . . 128
6.4 Code hints . . . 129
6.5 Code templates . . . 129
6.6 Code generation . . . 130
6.7 The iOS default keyboard . . . 131
6.8 The code entry keyboard (CEK) . . . 133
6.9 Resizing and docking in the CEK . . . 133
6.10 Keys for special characters in the CEK . . . 135
6.11 CEK keys with marking menus . . . 136
6.12 Marking menus for code templates . . . 138
6.13 Supportive widget for code template navigation . . . 138
6.14 Supportive widget for code completion . . . 139
6.15 *CEK study*: Demo task . . . 143
6.16 *CEK study*: Average keyboard frame and key size . . . 147
6.17 *CEK study*: Average key deviation . . . 148
6.18 *CEK study*: Key Deviation–Zoom Factor . . . 148
6.19 *CEK study*: Menu selections over time . . . 149
6.20 A revised key layout for the CEK . . . 157

6.21 *CEK simulation*: Applying the *Bayesian Touch Criterion* 159
6.22 A refined key interaction model for the CEK 160
6.23 Model for code completions in the CEK 165
6.24 A revised model for code entry 168

7.1 Properties for gesture-driven interaction 175
7.2 Examples of competing gestures in the editor viewport 176
7.3 *Touch IDE*: Optimized marking menus 179
7.4 *Touch IDE*: File browsing 181
7.5 *Touch IDE*: Working sets 183
7.6 *Touch IDE*: Editor layouts 184
7.7 *Touch IDE*: File sets 185
7.8 *Touch IDE*: File navigation gestures 187
7.9 *Touch IDE*: Interpolated scrolling and outline 189
7.10 *Touch IDE*: Cursor and context menu 190
7.11 *Touch IDE*: Selection handles and *selection rails* 191
7.12 *Touch IDE*: Selection context menu 191
7.13 *Touch IDE*: Zoomed editor layout and configured editor panes 193
7.14 *Touch IDE*: Error highlighting 194
7.15 *Touch IDE*: Code review 195

8.1 Components of an NUI code editing module 204
8.2 Service architecture in Eclipse Orion 208
8.3 Abstraction layer for a code editing subsystem 210
8.4 Exemplary IDE modules and submodules 213
8.5 Loose coupling and events 214
8.6 Asynchronous interaction of controllers and services 215
8.7 Examples of services and core objects 216
8.8 Adapters and bridges for two-way communication 217
8.9 MVC and commands 219
8.10 Custom gesture recognizers and conflict resolution 220
8.11 Concurrency and code analysis operations 222

9.1 Collaborative code reviews on a tabletops 232
9.2 Tangible exploration of code smells and refactorings 233

A.1 Pre-study questionnaire 261

A.2 Task descriptions 262
A.3 Post-study questionnaire 263

B.1 Handout in winter term 2012/13 267
B.2 Tasks in winter term 2012/13 270
B.3 Handout in summer term 2013 272
B.4 Tasks in summer term 2012/13 274

C.1 Pre-study questionnaire 277
C.2 Code entry task 278
C.3 Extended version of the code entry task 280

List of Tables

4.1 *RefactorPad*: Editor operations used in the study 75

5.1 Keyboard-driven cursor movement and selection commands 96
5.2 Mouse-driven selection commands 97
5.3 iOS selection gestures 99
5.4 Android selection gestures 99
5.5 Frequencies of triggered Eclipse commands 107
5.6 Selection directions 111

6.1 *CEK study*: Selection deviations for individual keys 149
6.2 *CEK study*: Selection frequencies for keys 150
6.3 *CEK study*: Relative frequencies for non-letter keys 151
6.4 *CEK study*: Frequencies and usage of code templates 152
6.5 *CEK simulation*: Code completion with predictions 166

Note on Writing Style

Before writing up my thesis, I consulted a book from Helen Sword [Swo12] for guidance on academic writing style. Sword's arguments (Chapter 4, "Voice and Echo") have encouraged me to reduce "passive or agentless constructions" in favor of writing in the first person. Although personal pronouns may not sound as objective at times, I hope this style more clearly communicates some of my thoughts and "readers can easily identify 'who's kicking whom'" (discussed by Richard Lanham in "Revising Prose", as cited in [Swo12]). While this work still frequently employs the passive voice for technical descriptions or descriptions of procedures, I have primarily applied the personal form in introductory sections, discussions, and conclusions.

Chapter 1

Introduction

In this chapter, I introduce the problem statement and the objectives of my work. Furthermore, I describe the research approach, outline the structure of this thesis, list related publications, and define basic terms.

1.1 Problem Statement

This work seeks to devise interaction techniques for editing textual source code on touch-enabled devices. Since devices with touchscreens have become ubiquitous, gestures and multi-touch interaction have received considerable attention in both research and development. Major software manufacturers have recently concentrated on mobile strategies and on improving their productivity tools for devices such as tablets. However, the field of programming and software engineering has only cautiously taken advantage of the interactive capabilities provided by modern hardware and software. The foundations of user interfaces for IDEs (Integrated Development Environments) and code editors have been built more than 40 years ago and seen only little change. In contrast, touch-enabled devices have rapidly been adopted by the masses and with their introduction, user interface paradigms have shifted from traditional desktop user interfaces, operated via mouse and keyboard, towards so-called *natural user interfaces*, operated via a touchscreen. Currently, no appropriate interaction techniques exist that allow programmers to efficiently edit textual source code–written in mainstream programming languages–on a touchscreen. My work aims at addressing this gap

through the design and evaluation of touch-centric methods for editing, selecting, and creating source code.

1.1.1 Challenges

The reasons for the slow adoption of touch-based interaction in the field of programming could be attributed to issues associated with the required hardware and software. Programming in mainstream languages typically implies efficient operation of a physical keyboard and executing keyboard shortcuts. Contrasting the familiarity and the tactile feedback of hardware keyboards, virtual keyboards of touchscreens have notoriously been slow and inaccurate. Moreover, popular touch-enabled devices such as smartphones and tablets restrict the available screen space and, despite continuous improvements, have not reached the computational power of desktop systems. These hardware limitations create challenges for rendering the complex user interfaces of development environments usable. In addition, software-related factors might have added to the slow adoption. Developers find themselves confronted by an over-abundance of features competing for their attention. IDEs, often grown over decades, generate substantial effort for porting these extensive feature sets to touch-enabled devices. The fundamental differences in hardware and user interaction might result in having to create entirely new tools that are tailored to the interaction models introduced by the “Post-PC Era”.

1.1.2 Prior Approaches

Prior research yielded projects that have enabled programming on touchscreens, but they have largely bypassed the issues of textual code input and editing. So far, researchers have primarily applied the following four strategies:

Visual Programming

Visual Programming (VP) replaces textual structures with graphical elements that programmers manipulate to specify program logic. While potentially well-suited for touchscreens, VP has not gained widespread adoption for classic software development scenarios and has mainly been utilized in specialized domains (e.g., audio-visual systems, mathematical environments, or programming for children). Frequently mentioned reasons for the limited success of VP are deficiencies in scaling to larger programs or problems with interoperability.

Syntax-directed Editing

Syntax-directed editing (also called *structure editing*) enforces certain constraints during editing in order to prevent errors and maintain the document structure. Similar to VP, syntax-directed editing may–precisely because of its restrictive nature–be well-suited for touch interaction but has not gained wide acceptance. Frequent criticism includes usability issues, caused by its various editing limitations, and the resulting inefficiencies. Although newer attempts could partially solve some of these issues, structure editing is still less flexible than free-form text editing.

Alternative Programming Languages

Prior work either resorted to alternate programming paradigms or created entirely new languages specifically for the purpose of improving the interaction on touch-based platforms. Particularly on space-constrained devices, languages with compact syntax offer advantages compared with widespread imperative and object-oriented programming languages. However, this approach either requires programmers to learn a new programming language or limits flexibility by enforcing artificial conventions.

Touchification

The term "touchification" [BHLD14] refers to running existing applications on touch-enabled hardware with little to no changes in their user interfaces. Touches on the screen are mapped to mouse coordinates that drive the interface elements. Since touches are more inaccurate than mouse pointers, individual elements may need to be enlarged or rearranged. Although this solution enables high reuse, IDEs require complementary techniques and workarounds to compensate for interaction issues that are rooted in the underlying WIMP paradigm.

It should be noted that existing work may not strictly fall into one of the categories but instead employ several strategies (e.g., alternative programming languages with structured and graphical components). However, none of the approaches enable flexible programming in mainstream programming languages while taking advantage of explicitly designed natural user interfaces. The efforts of this work are thus directed towards finding techniques that improve the interaction with source code on touchscreens *without* simplifying, restricting, or changing conventional programming.

1.1.3 Research Objectives

The primary research objective of this work is enhancing the textual editing capabilities of touch-enabled devices to be compatible with source code. The goal is not to reinvent code editing but instead to enable efficient interaction by taking the syntax and structure of source code into account. Since the text editing methods of conventional touch-based platforms are inappropriate for source code, this work aims at proposing interaction methods that compensate inefficiencies of typical text input and editing mechanisms. Usage scenarios include the ability to perform small-scale maintenance tasks on mobile devices such as tablets. The expected advantages and consequences of this approach are:

- The approach is largely language-agnostic and supports programming in mainstream languages.
- Users can reuse their existing skills for working with textual content.
- Developers can reuse the large infrastructure around textual representation of source code.
- The presented techniques are applicable *today* and do not assume any special requirements regarding hardware or device sensors.
- The techniques exploit the advantages of *direct manipulation* and multi-touch interaction; that is, instead of indirectly interacting via an attached device, users directly point at objects on the screen and perform gestures.

These general goals are divided into the following subgoals:

1. Understand how programmers *edit* and transform source code through gesture-driven interaction. The expected outcome are user-elicited gestures that serve as design guidelines or inspire novel interaction techniques.
2. Evaluate the mechanics of how programmers *select* code. The expected outcome are interaction techniques that increase the efficiency of selecting structural regions of source code.
3. Develop and evaluate methods that let programmers *create* new source code. The expected outcome are text entry methods that improve and simplify entering program syntax.

4. Design and implement the aforementioned subgoals, as well as supporting IDE modules. The expected outcomes are the design and software architecture for a coherent touch-enabled IDE.

1.2 Research Approach

Devising source code interaction methods for touchscreens requires an understanding of code editing in desktop environments, as well as the potentials and limitations of touch-based platforms. Consequently, this work considers previous work lying at the intersecting research areas of *Human-Computer Interaction* (HCI) and *Software Engineering* (SE). According to the ACM classification scheme[1], the relevant top-level categories are labeled "Human-centered computing" and "Software and its engineering", with the latter including software *tooling*. Myers [MK09] has stressed the increasing importance of software development tools incorporating findings from the area of HCI:

> "Many of the early work on software development tools was not useful (or at least not used) by professional developers, but in early 2000's, software engineering researchers started to take a more human-centered approach to the design and evaluation of these tools. [...] The common themes among these and similar examples is that studies of software development inform design, and evaluations of designs inform further study."

Although software development tools often include extensive feature sets, the evaluations of this work only cover the core phases of *code entry and editing*. Complementary tasks such as navigating code bases are separately addressed but have not been empirically evaluated within the scope of this work.

Evaluations and Limitations

The evaluations are carried out by means of user studies; that is, participants interact with a prototype and generate quantitative and qualitative data. The prototypes of the included studies consist of prepared code editing environments, both desktop- and touch-based. Quantitative data includes interaction events logged into a database for statistical analysis. Qualitative data is gathered through questionnaires where

[1] `http://dl.acm.org/ccs_flat.cfm`

participants indicate attributes relating to the user experience. Both sources are exploited to judge the suitability of an existing interaction technique or to propose new techniques based on the study results. I have employed both strategies in this work.

The first and last study (editing and creating code, respectively) were conducted in laboratory settings, while the second study (selecting code) was conducted in a realistic development scenario. The controlled nature of two of the studies affects the *external validity* and thus limits the generality of the results. However, the relative complex demands of the domain regarding prototypes and test setups have ruled out alternate evaluation forms. The participants of all studies have exclusively consisted of students with programming experience. On the one hand, studies with professionals might have generated results that more realistically reflect the behavior of programmers. On the other hand, professionals have vastly differing expertise, skill levels, and opinions concerning programming languages and development environments. In contrast, the individual differences among students may be smaller and hence result in more homogeneous user groups for the studies.

As far as the appropriate sample size is concerned, comparable studies in the field of HCI have often been based on the results of ten to twenty participants for within-subjects designs. The time and cost required for moderated testing usually prohibit larger sample sizes. The second study of this work could take advantage of a larger sample size (78 participants) since it was unmoderated and captured data through installed logging facilities. Overall, the results of this work should not be overgeneralized but rather be seen as means that have contributed to informed decisions about devised interaction methods.

1.3 Publications

This section contains a chronologically ordered list of publications, either directly arising from this work or from projects that are related to the topic. In each chapter, I will explicitly point out any content that is reused or not originally my own.

- Hartmut Glücker, Felix Raab, Florian Echtler, and Christian Wolff. EyeDE: Gaze-enhanced software development environments. In *Proceedings of the Extended Abstracts of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI EA '14, pages 1555–1560, New York, NY, USA, 2014. ACM

- Markus Fuchs, Markus Heckner, Felix Raab, and Wolff Christian. Monitoring students' mobile app coding behavior: Data analysis based on IDE and browser interaction logs. In *Proceedings of the 5th IEEE Global Engineering Education Conference*, Educon '14. IEEE, 2014
- Felix Raab, Christian Wolff, and Florian Echtler. RefactorPad: Editing source code on touchscreens. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 223–228, New York, NY, USA, 2013. ACM
- Felix Raab. CodeSmellExplorer: Tangible exploration of code smells and refactorings. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 261–262, Sept 2012
- Felix Raab, Markus Fuchs, and Christian Wolff. CodingDojo: Interactive slides with real-time feedback. In Harald Reiterer and Oliver Deussen, editors, *Mensch & Computer 2012 – Workshopband: interaktiv informiert – allgegenwärtig und allumfassend!?*, pages 525–528, München, 2012. Oldenbourg Verlag
- Felix Raab. Interaktionsdesign menschzentrierter Refactoring-Tools. *Information, Wissenschaft & Praxis*, 63(5):329–334, 2012
- Felix Raab. Collaborative code reviews on interactive surfaces. In *Proceedings of the 29th Annual European Conference on Cognitive Ergonomics*, ECCE '11, pages 263–264, New York, NY, USA, 2011. ACM

1.4 Structure

The rest of this work is divided into four main parts, organized as follows:

Part I: Background and Prior Work

Chapter 2: Introduces the context and background for subsequent chapters. The chapter covers fundamentals concerning source code and its forms of representation, and the relationship between programming and the concept of usability. Also, it describes typical components of software development environments and available tooling.

Chapter 3: Summarizes related research on touch-based development environments and interaction methods for code editors, including work from the field of text entry and editing, and gesture-driven execution of commands.

Part II: Source Code Interaction

Chapter 4: Introduces the types and details of typical code editing operations, including behavior-preserving structural transformations. Following that, the chapter reports the details and findings of a user study on code editing gestures and discusses implications for the design and implementation of gesture-driven code editors.

Chapter 5: Explains the mechanics of code selection by examining desktop and mobile editors. The chapter then reports the findings of a user study that has revealed selection patterns in a realistic software development situation. The final part includes the design of touch-enabled selection techniques.

Chapter 6: Introduces the challenges of touch-centric code input and presents the model and the design approach of a custom keyboard for code entry. Following that, the chapter reports the results of a user study centered around entering code with this custom keyboard. The final section presents a revised version of the keyboard and simulations of an enhanced model for code entry.

Part III: Design and Implementation

Chapter 7: Presents the design of a coherent system integrating the studied interaction techniques for touch-centric code editing and their supporting IDE modules. The chapter introduces concrete IDE modules and explains the rationale behind all design decisions.

Chapter 8: Discusses technical aspects and software architecture. The chapter details principles and patterns used for implementation and describes the concrete architecture of selected sub-systems, their communication mechanisms, and technical constraints.

Part IV: Conclusions

Chapter 9: Summarizes the essence of previous chapters and highlights the research contributions. The second part of this chapter attempts to identify opportunities for future research directions and introduces related projects that have emerged from this work.

1.5 HCI Terms

This section briefly defines basic terms from the field of Human-Computer Interaction that are repeatedly used throughout this work. Terms related to source code editing and tooling are separately introduced in each of the respective chapters.

Direct Manipulation
Originally introduced by Shneiderman [Shn83] to describe an *interaction style* that lets users manipulate visual object representations on a desktop system. Users point at objects to trigger actions and receive instant feedback. Alternate interaction styles are command language or menu selection.

Natural User Interface (NUI)
Having been referred to as "marketing name" [Nor10], the term may today loosely describe the interface and interaction style of touch-enabled platforms: Users instantly interact through touch interaction and gestures, as opposed to operating a keyboard and mouse. Ideally, an NUI reduces the learning cost and guides the user in transitioning from novice to expert [Wig10].

Windows Icons Menus Pointer (WIMP)
Could be regarded as opposing interaction style to NUIs (see above); that is, conventional desktop systems with graphical user interfaces that are operated via keyboard and mouse. Most actions are triggered by pointing to objects or executing menu commands.

User Experience (UX)
The concept of "user experience" is commonly understood to describe the entire experience of how users perceive a product. This experience may include aspects such as the usefulness of the product, its usability (see Chapter 2), or the provoked emotions. ISO 9241-210 defines UX as "a person's perceptions and responses that result from the use or anticipated use of a product, system or service"[2].

[2] `http://en.wikipedia.org/wiki/User_experience`

Part I

Background and Prior Work

Chapter 2

Background

In this chapter, I introduce the context and background for subsequent chapters. The first part addresses fundamentals concerning source code and its forms of representation, programming languages, and the relationship between programming and the concept of *usability*. The second part describes the typical components of software development environments, highlights the different types of available tools, and introduces the challenges of enabling programming on devices with touchscreens.

2.1 Source Code, Programming, and Usability

The Oxford Dictionary of English defines source code as "a text listing of commands to be compiled or assembled into an executable computer program." Members of the conference SCAM (Source Code Analysis and Manipulation) have agreed upon the following definition of source code [Har10]:

> "For the purpose of clarity 'source code' is taken to mean any fully executable description of a software system. It is therefore so construed as to include machine code, very high level languages and executable graphical representations of systems."

While the Oxford definition explicitly states that source code consists of text, the SCAM definition does not explicitly mention text but instead emphasizes the aspect of execution. Source code is usually created as result of *programming*. Steele [Ste98],

designer of multiple programming languages, has defined the verb "to program" as follows:

> "To program is to make up a list of things to do and choices to make, to be done by a computer. Such a list is called a program."

Programming has also been defined as "the act of assembling a set of symbols representing computational actions" [KP05] or "the process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer" [MK09]. The authors of source code are usually called *programmers* or, more generally, *software developers*. Programmers create source code in order to create programs for end users, although often external programs become the actual users before humans operate the final programs. Consequently, programmers are themselves users who *use* source code to solve their tasks. In any environment where users need to solve tasks, the concept of *usability* is accepted to be important. The ISO 9241 definition of usability reads:

> "The effectiveness, efficiency and satisfaction with which specified users achieve specified goals in particular environments."

Therefore, programmers ("specified users") should be effective, efficient and satisfied with creating source code or programs ("specified goals"). The term "particular environments" could be interpreted as referring to source code itself as the environment or–since source code is usually not created using pen and paper–to the tools used to create source code. I will discuss both interpretations (language-level and tool-level) in the following sections.

2.1.1 Programming Paradigms and Languages

When source code is viewed as the environment for the programmer, the *actual* environment is not source code itself (or the instructions for the machine to execute) but the programming language and its rules. Today, a large number of programming languages exist and each language serves different purposes. When writing code, the programmer has to adhere to rules defined by the programming language. Moreover, programming paradigms are, by nature of their definition, fundamentally different. As a consequence, the programmer's choice to use one or more languages for solving his task directly affects how effective, efficient, and satisfied specified goals can be solved.

Although the question of which language design or programming paradigm is most efficient for particular tasks is not the primary concern of my work, it is worth noting that different languages or paradigms also require different tooling. For example, the source code of some symbolic programming languages may include non-textual components that depend on different forms of editing (and therefore tooling) than regular text. Code could contain a mathematical formula, an image, or an interactive widget. The overarching programming paradigm dictates, to some extent, which representational forms other than plain text are adequate. Van Roy [VR09] defines the term “programming paradigm” as “an approach to programming a computer based on a mathematical theory or a coherent set of principles”. He argues that an ideal programming language should support multiple paradigms so that programmers are capable of solving a variety of problems; however, this “multiparadigm programming” gives rise to questions on how different paradigms are best supported within a single environment so as to reduce the cognitive load for programmers.

In addition, the creators of programming languages often seem to make arbitrary choices as far as syntax and semantics are concerned. As a result, the design of languages may be based more on personal taste of a small group of developers than on objective measures [HLG12]. This begs the question whether an objective measure on what exactly constitutes a “better” programming language exists. Stefik et al. [SSSS11] have created a programming language that is based on empirical metrics gained from a long-term study. They claim that results from their study towards more intuitiveness in programming languages were “highly encouraging”. For example, they found that participants rated the syntactical keyword “repeat” significantly more intuitive as the widely used keyword “for”. In another study [HGL13], researchers remarked, “physical aspects of notation, often considered superficial, can have a profound impact on performance”. Although these results appear intriguing, it seems unlikely that they will have a major influence on mainstream programming languages currently used in industry. Myers and Ko [MPK04], for instance, note:

> “It is somewhat surprising that in spite of over 30 years of research in the areas of empirical studies of programmers (ESP) and human-computer interaction (HCI), the designs of new programming languages and debugging tools have generally not taken advantage of what has been discovered.”

The adoption of programming languages depends on a variety of factors. In a recent and extensive study, Meyerovich and Rabkin [MR13] analyzed a large number of

project repositories and surveys of programmers in order to empirically find out which factors contribute to the popularity of programming languages. The authors report three findings: First, they observe a power-law distribution in language adoption (few languages are used the most). Second, external factors (e.g., available libraries, existing code, and personal experience) are more important than intrinsic language features (e.g., performance, reliability, semantics). Third, developers adopt and abandon languages multiple times during their professional life, independent of age. Furthermore, they tend to be familiar with more languages when teachers introduced them to different language paradigms in their education. Finally, Meyerovich and Rabkin remark, "developers consider ease and flexibility as more important than correctness".

2.1.2 Cognitive Aspects of Programming

The gap between industry and academia becomes even more evident in research on the psychology of programming, specifically on cognitive architectures and mental models. According to Hansen et al. [HLG12], the first period of research on the cognitive aspects of programming began in the 1960s and 1970s. Researchers applied psychological theories to computer science and carried out experiments that "looked for correlations between task performance and language/human factors – e.g., the presence or absence of language features, years of experience, and answers to code comprehension questionnaires". Hansen et al. explain that the second period, starting at 1980, has focused on cognitive models in order "to explain basic mental processes and their interactions". In other words, they have investigated the usability of programming languages.

As previously mentioned, some studies have had intriguing results but dominant programming languages have ignored academic findings. It is remarkable that scientific accomplishments have not shaped programming more fundamentally, not least since it is often referred to as one of the mentally most demanding tasks humans perform. A characteristic example of this omission, mentioned in [HLG12], was the decision on which design of two versions of a certain language feature in the C++ language ("Concepts") to reject. After debates among disagreeing experts, the feature was finally removed without considering an objective study of the advantages and disadvantages of those designs.

The differences in mental models between novice- and expert-programmers are well known. Winslow [Win96] reviewed significant findings of psychological studies about computer programming in a paper about "Programming Pedagogy". He notes, "[novices] lack an adequate mental model of the area" and "it takes approximately ten years to turn a novice into an expert". A detailed discussion of mental models and cognitive architectures for programming is beyond the scope of my work. However, this section intends to acknowledge tooling (the focus of this work) as only *one* vital component for improving the usability of programming. The Programming paradigm, the design of a programming language, and mental models positively affect if programmers feel that working with source code is "usable" according to the ISO 9241 definition.

2.1.3 Forms of Representation

The definitions of "Source Code" at the beginning of the chapter do not unequivocally state that source code has to be text. I would argue that most programmers–other than those working in highly specialized domains–usually think of source code as consisting of "listings of commands" as in the Oxford definition. Code, however, might also be represented in purely visual ways or textual and visual ways (*hybrid* representation). Here, the word "representation" refers to the *external* representation (i.e., the output that the programmer interacts with on the screen), and *not* the *internal* representation. (Visual representations are often automatically translated to textual representation without programmer intervention.) I will introduce the aforementioned three forms of external representation in this section since they arguably belong to the most deciding factors concerning how programmers interact with source code. These forms also differ significantly in how toolmakers should design appropriate editors.

Textual Representation

At the present time, textual representation is evidently the most widely used form, not least because programming was invented as sequences of text-based commands. While devices using punch cards could already perform calculations purely mechanically in the 18th century, the first electronic computer was built in the 1940s [Rob08]. The act of programming involved using *assembly language* to write low-level instructions for the machine to execute. Today, *programming* usually means using a high-level

programming language, such as Java or C, and entering textual instructions into an editor. Most editors colorize parts of the text according to the API and semantics of the programming language, thereby helping programmers to recognize keywords and relationships in the code.

Depending on the editor, applications support programmers to varying degrees in writing code; however, textual programming requires the programmer first to learn and internalize the programming language. To be able to solve different tasks efficiently, programmers should usually learn more than one programming paradigm [VR09], which also often implies learning more than one programming language. This text-based nature of most programming languages provides an opportunity for usability improvements of programming: All improvements that affect the general interaction with text directly apply to text-based programming environments. (The same is not necessarily true for visual programming, for instance.)

Visual Representation

Visual representations could be divided into two categories: Representations that solely act as a "facade" to an underlying text-based programming language, and representations that were primarily designed to be visual. The latter could internally be translated into a text-based representation; however, the programmer does not see or modify the output in that case. Visual representation means programming concepts are not exclusively expressed as text but as visual elements that the programmer manipulates in order to create a meaningful program. Components could be user interface elements such as boxes, sliders, arrows, or any other graphical elements, including text snippets displayed in input fields. The editing activity typically consists of configuring, rearranging, and connecting components on a canvas, but the details depend on the concrete development environment.

Compared with text-based languages, where any text editor can be used to modify programs, visual programming languages and their tight coupling to the editing environment could be regarded as disadvantage. Some domains, however, derive particular benefits from the graphical representation. For instance, Petri nets, a modeling language for distributed systems, communicates concepts through its graphical notation and may thus increase the programmer's understanding of the overall system. Moreover, user interfaces for creative audio and video tools are well suited because the development environments give artists visual real-time control and feedback over

music, video, or hardware installations. This degree of interactive and intuitive program manipulation is hard to imagine with text-based languages. Besides artists, novice-programmers are a popular target audience for environments with visual representation. Programming concepts can be simplified through visual notation, which benefits novices in learning the fundamentals of programming. Such environments often fall into the previously mentioned category of tools that purely act as a "facade" for code generation.

According to critics, visual programming could not deliver its promise of increasing code understanding [MK09]. They point out that research has shown text to be more natural and space-efficient than visual languages [MK09]. Other mentioned disadvantages include the lack of specifications and portability, difficulties with automatically laying out the components of a program without causing disorientation, or the challenges involved in creating the same infrastructure and tools that currently exist for text-based languages [Mye90].

Hybrid Representation

The debate on whether the future of programming is textual or visual in nature has been persistent since the development of the first visual languages. Since arguments in favor of both approaches could be listed, "the best of both worlds" might be combined in *hybrid* representations. *Symbolic programming* in particular appears as a natural fit for representations consisting of mixed modes. For example, a symbol could be a mathematical formula rendered in mathematical notation instead of a sequence of characters. Users could then manipulate the formula using a widget that provides unique capabilities for maths, while other parts of the program remain textual. The potential of symbolic programming goes far beyond this simple example. Recent projects such as the "Wolfram Language"[1] have demonstrated the vast possibilities that emerge when symbolic manipulation is linked with built-in knowledge about computation and artificial intelligence.

Discussion

In textual development environments, programmers can reuse all of their already acquired skills for working with text. Text allows programmers to express their

[1] `https://www.wolfram.com/language/`

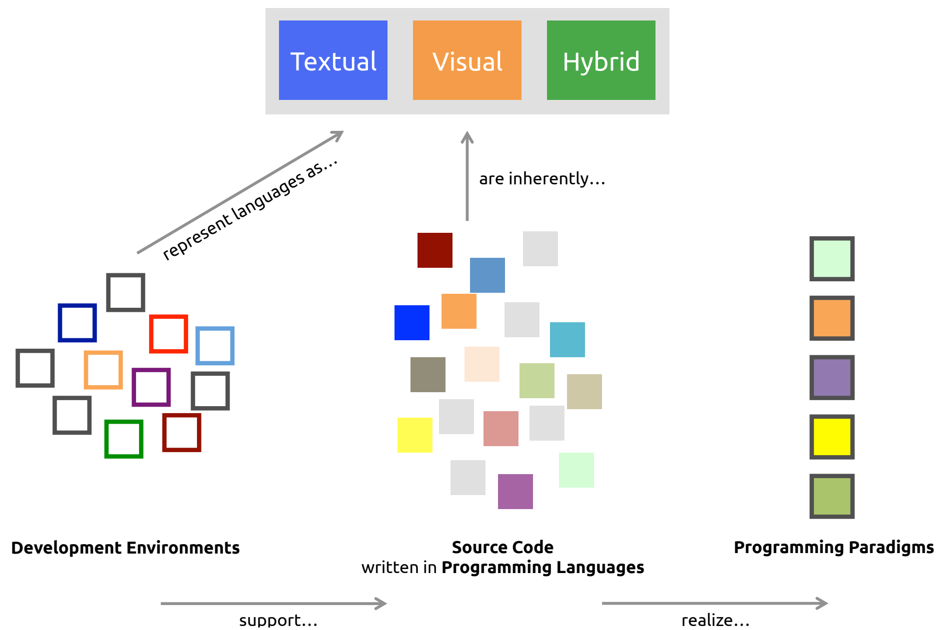


Figure 2.1: Relationships between programming languages, programming paradigms, development environments, and forms of representation. (Parts of this diagram are inspired by Figure 1 in [VR09].)

intentions precisely; files with textual content can be easily stored and exchanged, or searched and compared. Furthermore, the large portion of existing source code is text-based, requiring maintenance work for decades to come.

Despite decades of research and hundreds of published papers, visual programming still has not gained significant acceptance. Currently, its strength primarily lies in specialized domains. Although some of the existing programming paradigms lend themselves better to visual programming than others (e.g., the dataflow and reactive programming paradigm), successful visual programming may require the introduction of new programming paradigms that are *inherently* visual and exploit new interactive capabilities. Mathematical and scientific computing can take advantage of hybrid representations that visualize the output of computations within the source code. As programming languages further advance, hybrid forms and embedded widgets might increasingly appear in mainstream development environments. (Hybrid approaches are further discussed in the following sections on types of IDEs.)

Figure 2.1 illustrates the relationships between programming languages, programming paradigms, development environments, and forms of representation: Source code is

written in different programming languages that realize one or more programming paradigms. Languages are either inherently textual, visual, or hybrid. Development environments, in turn, support one or many languages and represent languages either as textual, visual, or hybrid. (Note the difference between *inherently* and *represent*.)

2.2 Integrated Development Environments

Programmers typically create and edit source code in *Integrated Development Environments (IDEs)*; that is, applications that provide collections of tools for working with code. Some programmers favor *text editors* over IDEs. Although text editors are components of IDEs, as I will discuss later, the boundaries between an IDE and a text editor can be blurred.

2.2.1 IDE Components

To my knowledge, an official definition of IDE does not exist; nevertheless the term commonly implies presence of typical features for editing, inspecting, running, and debugging source code. The *visible* feature set of IDEs could be divided into the following functional units:

File Browsing and Version Control
Source code is usually browsed in hierarchical views of files and folders, as known from the file browsing facilities of operating systems. In addition, VCS (Version Control Systems) may link to the file browser and–depending on the degree of integration–enable features for interacting with code repositories.

Projects and File Sets
An IDE allows programmers to create a project that groups related source code as form of organization. File sets are organizational structures for creating sub-groups of source code within a project. The term for this concept varies and an IDE might have multiple mechanisms for sub-grouping and code organization or none at all. Projects may also persist settings for the current session and restore its state when the programmer reopens the IDE.

Code Editing and Code Intelligence

Code editing in a text editor is one of the core capabilities of an IDE. The editing facilities typically include syntax and error highlighting, auto-completion, code formatting, bracket matching, code folding, interactive gutters, help and documentation lookup, and other forms of instant feedback for programmers. Features concerning code intelligence (syntactic and semantic analysis) considerably vary between IDEs. The availability of automatic program transformations (*Refactoring*) is a distinctive feature that sets IDEs apart from pure text editors. In addition to providing general tooling for programming, most IDEs support multiple programming languages. Syntax highlighting, code completion, or error reporting, for example, are language-specific features. Some IDEs support only a single language or a subset of related languages (e.g., web programming languages or languages for mobile development).

Navigation and Search

Since larger projects may consist of hundreds or thousands of files, navigating and searching source code is a crucial IDE feature. Code navigation can take various forms. IDEs provide facilities for navigating within a single source file (*intra-code* navigation) and navigating between multiple source files (*inter-code* navigation). Programmers navigate source code by selecting linked keywords or entities in hierarchical views. Search commonly allows users to find and replace code within individual files or project-wide.

Testing and Debugging

The process of testing or executing programs differs between programming languages, but IDEs typically allow code to be run and tested in predefined or configurable target environments. Users can attach a debugger to running programs for diagnosing and finding errors.

Extensibility and Customization

Most IDEs provide an architecture for *plug-ins* to let users install additional features on the environment. Features could be enhancements to the user interface or support for new programming languages. Standard customization options include the possibility to change the editor appearance, keyboard shortcuts, or compiler settings.

2.2.2 Usability and Usefulness of IDEs

The high number of IDE modules indicates that programmers are confronted with a considerable amount of features. Over the years, IDEs have grown in functionality and tend to keep adding more components. The abundance of functionality has led to convoluted user interfaces and an enormous number of keyboard shortcuts. Perhaps unsurprisingly, a high number of changes to Eclipse, a popular open-source IDE, can be attributed to usability-related issues [HW09].

In their review of literature on IDE usability, Kline and Seffah [KS05] note:

> "All of these results indicate that, in general, IDEs and related tools have not had the positive effects on developer productivity and software quality anticipated in the early 1980s. This is unfortunate because the cost of adopting an IDE is not insignificant: It can be as high as about $20,000 (US) per developer after all product and training costs are considered (Lending and Chervany, 1998)."

The rise of polyglot programming (i.e., programming in multiple languages) has forced developers to use more than one development environment. The requirement to create programs for several target platforms, as it is common practice for mobile development, entails that developers become familiar with different IDEs. This switching of development tools can lead to productivity loss because the user interface, keyboard shortcuts, and configuration are not consistent between applications.

Kline and Seffah also stress the significant difference between usability and usefulness. In their paper "Designing Useful Tools for Developers" [LM11], LaToza and Myers argue, "useful tools must solve an important problem". They define an important problem as one that sufficiently satisfies the criteria of frequency, duration, and quality impact. For example, an issue that frequently occurs with little impact and an issue that less frequently occurs with high impact could both be regarded as "important problems". In Chapter 3, I present work for solving such problems.

2.2.3 Text Editors and IDEs

Usability issues and confusing interfaces might have contributed to some developers favoring text editors over IDEs. Text editors, as standalone applications, differ from IDEs in that they lack particular components and graphical tools of IDEs but

instead provide advanced text editing features. Since most of these operations are completely keyboard-driven, users can reach high efficiency once they master the keyboard shortcuts. Some text editors offer extensions points to support multiple programming languages or third-party plug-ins that add specific functionality. When such extension mechanisms are exploited, text editors begin to resemble IDEs. Although the boundaries between an IDE and a text editor can be blurred, developers seem to have a clear idea of which concrete applications belong to each type.

The popular open-source text editor Emacs[2] is an example of a text editor that–through its extension architecture and customization options–can be turned into a full-fledged IDE. "Emacs-type" editors [Fin91] have arguably had major influence on modern IDEs. As a consequence of the number of available tools, developers regularly engage in debates[3] about the advantages of their chosen environment and point out the weaknesses of competing products. Although these comparisons are rarely based on objective evidence, there are empirical studies discussing trade-offs between IDEs and text editors.

For instance, Dillon et al. [DAB12] conducted a study in a python programming course where one group of students had transitioned from using an IDE to using a text editor, while the other group had transitioned from text editor to IDE usage. (The authors refer to IDE as "visual environment", namely IDLE, and to text editor as "command-line environment", namely VIM.) Dillon et al. summarize, "the consistency and affordance of certain features in visual environments could cause novices to develop a false perception of programming"; and, in contrast, "command line environments may enable novices to develop better mental models for programming because of their limited features, which could also allow them to transition to other environments much easier". Studying expert programmers in this way, however, may be difficult due to a variety of confounding factors.

2.3 Types of IDEs

Very recently, the programming language BASIC has celebrated its 50th birthday. BASIC played an important part in the development of the first application that could be labeled as IDE. In the following sections, I present illustrative examples of historic tools,

[2] `http://www.gnu.org/software/emacs/`
[3] `http://en.wikipedia.org/wiki/Editor_war`

introduce different types of IDEs, and conclude with recent developments in the IDE landscape. Most of the examples are either open-source or commercial applications, or applications that have grown out of research projects. More research-oriented projects and enhancements to individual modules are presented in Chapter 3.

2.3.1 The first IDE

The Dartmouth Time Sharing System (DTTS) revolutionized computing in the 1960s since it enabled multiple users to operate a computer via terminal. In his paper on BASIC [Kur81], Thomas E. Kurtz explains that, before DTTS, executing programs involved punch cards, printouts, and long waiting times until users received the results of their programs. The introduction of DTTS and BASIC eliminated administrative issues and was better suited for teaching students the fundamentals of programming. Students could interact with their programs in an environment that, in a basic form, resembled an IDE. Kurtz notes:

> "[...] the user deals directly only with his BASIC program. He need not even know that such things as "object code" exist. The user could compile (by typing RUN), receive error messages, edit by typing line-numbered lines, and recompile, all within seconds."

2.3.2 Textual Environments

As previously mentioned, text editors (source code editors) become IDEs through their extension architecture. Emacs (Figure 2.2a), originally developed by Richard Stallman in the 1970s, includes a basic graphical user interface that could be operated via the mouse. However, the efficieny commonly associated with operating text editors is gained by manipulating text through keyboard shortcuts or composed keybindings. Other popular text editors for code editing include VIM[4] or Sublime Text[5]. VIM assumes a special position through its modal editing style and composable editing operations. While general-purpose editors let users flexibly edit textual content, *structure editors* impose constraints during editing. The *Cornell Program Synthesizer* [TR81] is an early example of an editor enforcing syntax-directed editing and thus trading flexibility for

[4] `http://www.vim.org/`
[5] `http://http://www.sublimetext.com/`

maintaining the document structure. In chapters 3 and 5, I will revisit text editors and their different interaction styles.

(a) Emacs[6]

(b) Vim[7]

Figure 2.2: Two popular text editors: Emacs and Vim.

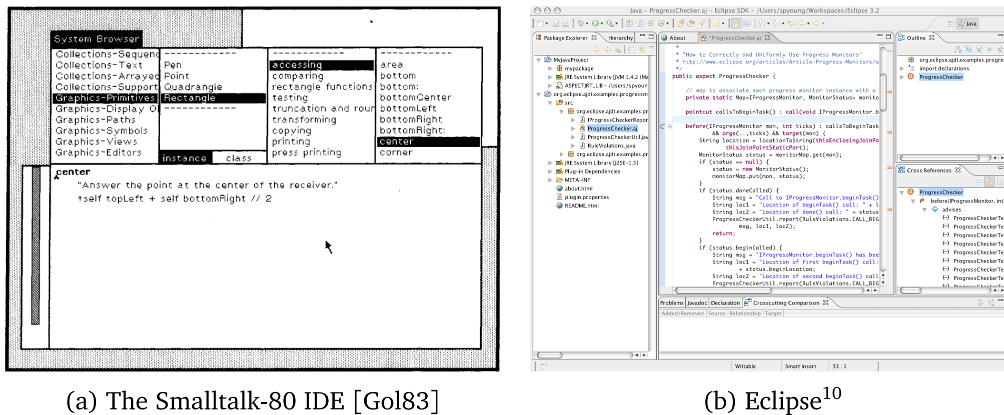
REPLs (Read-Eval-Print-Loop) are possibly the simplest form of text-based environments. An REPL could be a UNIX command line shell that reads user input, evaluates the input, and prints the output before returning back to the read state. Consequently, REPLs could be regarded as rudimentary text-based development environments.

2.3.3 Modern Desktop Environments

Although almost all IDEs can be labeled as "Desktop IDEs", by this term, I refer to conventional IDEs that today many programmers use for classic software development tasks. The IDE for Smalltalk, one of the first object-oriented languages, preceded modern software development environments for desktop computers. The Smalltalk IDE has been described as follows:

> "[...] first true Integrated Development Environment (IDE), and the first IDE that eliminated the "Edit-Compile-Link-Run-Test-Debug" build cycle by integrating them all together so that cycle turn around time can be

[7] http://upload.wikimedia.org/wikipedia/commons/5/5a/Emacs-screenshot.png
[7] http://media.cdn.ubuntu-de.org/wiki/attachments/11/39/vim-in-action.png



(a) The Smalltalk-80 IDE [Gol83]

(b) Eclipse[10]

Figure 2.3: The Smalltalk IDE as predecessor to modern IDEs such as Eclipse.

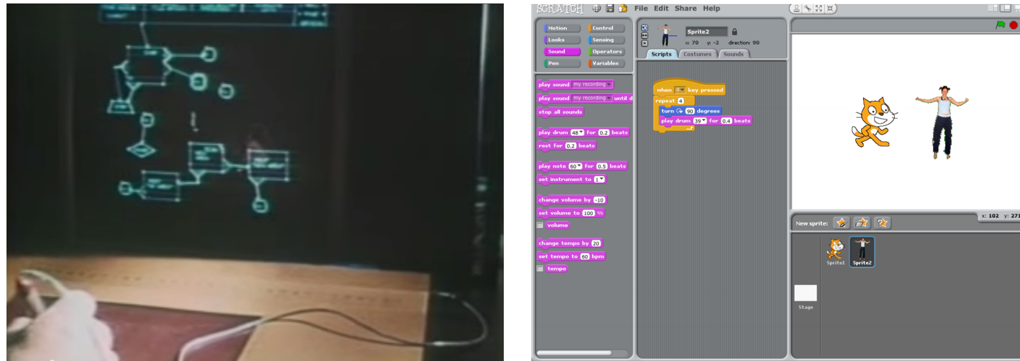
> measured in seconds and all phases can be active at once - true interactive development![8]”

Figure 2.3 shows the early user interface of the Smalltalk-80 IDE. The system browser on the top allowed programmers to navigate through categories and to easier find programmable objects, methods, and properties; the scripting pane below displays modifiable code. Although modern IDEs have extended navigation and browsing facilities, the Smalltalk browser could be regarded as a model for the *class browsers* of modern IDEs.

Eclipse (Figure 2.3b), a project initiated in 2001 and promoted by a consortium of industry leaders[9], is a widely used open-source IDE. Its open and extensible architecture has been a “catalyst” [MK09] for a number of tool-focused software engineering studies that otherwise would have been difficult and costly to conduct [MK09].

Other popular IDEs include Netbeans (open-source)[11], IntelliJ IDEA (commercial)[12], Microsoft’s Visual Studio (commercial)[13] or Apple’s Xcode (free but closed-source)[14]. Some IDEs (e.g., Visual Studio or Xcode) primarily target the environment and ecosystem of the manufacturer’s own programming languages.

[8] `http://www.smalltalk.org/articles/article_20040000_11.html`
[9] `http://www.eclipse.org/org/`
[10] `http://www.eclipse.org/screenshots/images/AJDT-Mac.png`
[11] `http://netbeans.org/`
[12] `http://www.jetbrains.com/idea/`
[13] `http://www.visualstudio.com/`
[14] `http://developer.apple.com/xcode/`



(a) Rand Corporation's GRaIL[16]   (b) MIT's Scratch[17]

Figure 2.4: Two environments for visual programming with almost 40 years between their development: GRaIL and Scratch.

2.3.4 Visual Programming Environments

According to Myers [Mye90], "Visual Programming" "[...] refers to any system that allows the user to specify a program in a two (or more) dimensional fashion". He emphasizes the difference between "Visual Programming" (VP) and "Program Visualization" (PV): While the first term denotes programs created using graphics, the latter implies textually specified programs, with some parts of the program being visualized after creation. Myers uses the term "Visual Languages" (VL) to refer to both VP and PV and, in his taxonomy, further differentiates between compiled and interpretive languages.

Remarkably, one of the first environments for graphical programming was already developed in 1968 and still appears futuristic when compared to the state-of-the-art of current development environments. The project GRaIL (GRaphical Input Language, Figure 2.4a) provided programmers with an editor that was operated using a tablet and a stylus. The system recognized drawn forms and converted them to parts of a flowchart. GraIL interpreted all hand-drawn figures and stylus gestures in real-time while showing the result on a display surface. The details of the interaction are described in a research memorandum published by Rand Corporation[15].

[15] http://www.rand.org/content/dam/rand/pubs/research_memoranda/2005/RM5999.pdf

[17] http://www.youtube.com/watch?v=QQhVQ1UG6aM

[17] http://scratched.media.mit.edu/sites/default/files/GettingStartedGuidev14.pdf

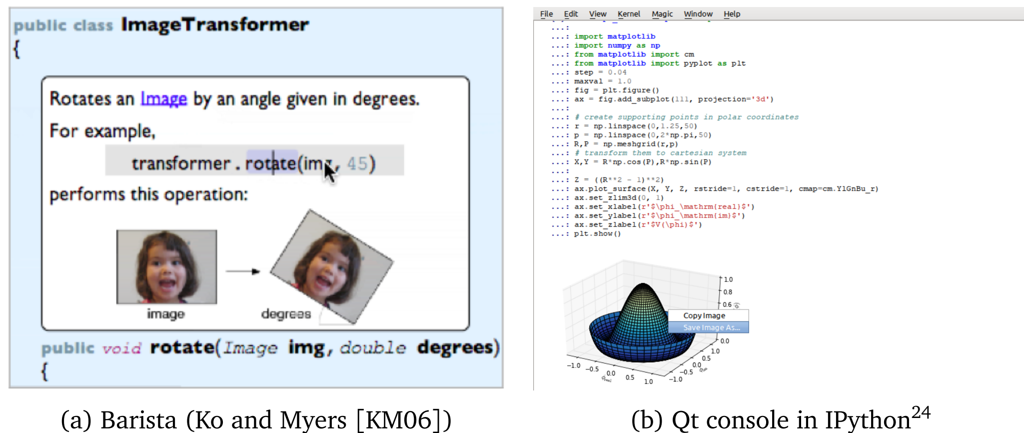
MIT's *Scratch* (Figure 2.4b), developed much later in 2006, is an example of a visual programming environment aimed at children and youths. Users construct programs by arranging and configuring code blocks on a canvas. Although many other visual environments exist, Scratch stands out since it has been subject of a considerable amount of research[18] around visual programming. *Android App Inventor*[19], another MIT project, is similar to *Scratch* in its user interface concepts but targets mobile application development. An extensive review of programming environments and languages for novice programmers can be found in a publication by Kelleher and Pausch [KP05].

In contrast to development environments for novices, *domain-specific* visual environments are used by experts in highly specialized areas, such as scientific modeling and engineering, design of electronic circuits, algorithmic trading, or 3D modeling. For creative audio and video applications, artists frequently work with *flow-based* programming environments. For instance, in the environment *vvvv*[20], the effects of visual manipulations occur in real-time when component parameters or links between elements are changed. This "wiring together" of components gives programmers immediate feedback and thus allows them to interactively refine their compositions. The scientific modeling environment *LabVIEW*[21] uses similar techniques of graphically connecting functional blocks and thereby controlling the flow of data through the program.

2.3.5 Hybrid Environments

Classifying environments into textual, visual, and hybrid environments leads to a fine line to be drawn between visual programming environments and *hybrid* environments. Depending on the definition, an environment allowing programmers to specify scripts containing program logic as part of an otherwise graphical specification of a program, could be classified as visual environment; here, this approach is referred to as "hybrid" environments. Apple's *HyperCard* [Goo87], released in 1987 and frequently mentioned as predecessor of modern authoring tools or visual GUI builders, may fall into this category. The development environment allowed users to attach scripts to objects arranged on virtual stacks of cards. Although the application primarily targeted

[18] `http://scratch.mit.edu/info/research/`
[19] `http://appinventor.mit.edu/`
[20] `http://vvvv.org/`
[21] `http://www.ni.com/labview/`



(a) Barista (Ko and Myers [KM06])

(b) Qt console in IPython[24]

Figure 2.5: Two hybrid development environments mixing textual and graphical elements.

non-programmers, the scripts controlled the appearance and interactivity; it thus represented a hybrid approach to programming.

The research project *Barista* (Basic Abstractions for Rapidly Implementing Structured Text-editing Applications) by Ko and Myers [KM06] is a framework for building hybrid code editors. Figure 2.5a shows the user interface of a *Barista* application displaying text alongside graphical content in the editor window. The authors call this example the "media-rich annotation of a Java method". The textual source code remains editable as in a purely textual environment, but additional embedded elements represent individual parts of the code visually and interactively (labeled "situated, task-appropriate views" by the authors).

IPython[22], another project supporting mixed presentation forms, lets developers interactively work in the terminal or a web-based "notebook". IPython describes its notebook as "a web-based interactive computational environment where you can combine code execution, text, mathematics, plots and rich media into a single document"[23]. The architecture of IPython is *language-agnostic*; that is, its kernel can be reused for other programming languages that aim to facilitate interactive scientific computing. Figure 2.5b shows the output of a graphics operation in IPython's Qt console as inline plot below the corresponding textual setup code and its final method call for plotting the result.

[22] http://www.ipython.org
[23] http://ipython.org/notebook.html
[24] http://ipython.org/_static/screenshots/ipython-qtconsole-thumb.png

2.3.6 Recent Developments

Recently, two types of development environments have received new interest, namely *live-programming* environments and *Web IDEs*. I have briefly mentioned the concept of live-programming in visual environments for audio and video applications. However, live-programming is also applicable in textual environments, as demonstrated by McDirmid [McD13]. He argues that some live-programming environments are only partially useful since they are limited to showing the visual result, whereas steps in between single calculations (the program execution) remain hidden. With his language *YinYan*, he attempts to fix the issue by allowing programmers to "probe" selected expressions that show their values in the source code during the execution of the program. Additionally, YinYang's "tracing" enhances regular print-debugging by linking navigable output back to source code positions. These concepts endeavor to achieve "true" live-programming environments where editing, debugging, and deploying applications occur in real-time without technology-imposed interruptions and delays while programming.

Successfully crowd-funded projects such as *LightTable*[25] indicate that instant feedback attracts wide interest beyond the research community. This seems to be further supported by Apple, who have recently integrated a feature called *Playgrounds* into the Xcode IDE. *Playgrounds* are enabled by Apple's new programming language *Swift*[26] and allow developers to edit code and immediately see the results of execution. When code runs over time, additional panels visualize the changing values and let users "scrub" through the code so that single execution steps can be inspected and fine-tuned.

Web IDEs, as the name suggests, are IDEs running within web browsers instead of regular application windows. Since applications from a growing number of domains are brought to web browsers, Web IDEs may be the logical next step in IDE evolution. The availability of open-source components for basic code editing might have contributed to the steady increase of new web-based development environments. Organizations behind large IDEs such as Eclipse or Visual Studio, have been working on web-based code editors (e.g., Eclipse Orion[27]) or on moving individual components to the web (e.g., Visual Studio Online[28]). According to Kats et al. [KVKV12], Web IDEs enable "connectedness", "centralized configuration and deployment", "integration with other

[25] http://www.lighttable.com/
[26] https://developer.apple.com/swift/
[27] http://www.eclipse.org/orion
[28] http://www.visualstudio.com/

services", and "infinitely scalable resources". However, Kats et al. note that IDE architecture must be fundamentally re-examined since the web platform imposes numerous technical and social constraints. The authors' outlined research agenda shows that moving software development to the web generates a number of challenges for future research.

2.4 Programming on Touchscreens

In the preceding sections of this chapter, I have presented definitions of the term "source code" and given an overview of programming languages, paradigms, and usability aspects. I have introduced different forms of representation and types of IDEs for working with these representations. While future IDEs might implement more live-programming features, run on the web, or realize hybrid programming paradigms, the previous discussion has so far not considered another important factor, namely the target device. Except for few systems, such as GRaIL, all shown IDEs run in desktop environments. The term *WIMP* (Windows Icons Menus Pointer) has commonly been used to refer to the dominant interaction style of conventional dektop systems: Users interact, through the mouse and keyboard, by selecting icons and menu entries on a window-based operation system. In recent years, however, more and more "Post-WIMP" systems have entered the market. A large proportion of smartphones and tablets use touchscreens that require different user interfaces and interaction styles.

While there is some research on touch-based programming, major IDEs have yet to be adapted to work on touchscreens. The cautious adoption of new target devices and the general stagnation of improvements in user interaction could be attributed to various factors. On the one hand, IDEs have often been developed over several years (or decades) and thus tend to be complex with regard to their internal application architecture and their external user interfaces. This inherent complexity generates challenges for porting the systems to new platforms. IDEs might need to be entirely re-architected and re-designed for space-constrained touch-enabled devices. On the other hand, in order to edit source code and handle the large amount of functionality, efficient usage of desktop IDEs relies on the keyboard. Text input and the absence of keyboard shortcuts, in contrast, has been a well-known shortcoming of devices with touchscreens. Despite considerable research in the area of touch-based text entry,

almost all commercial systems provide software keyboards that can be difficult to use.

Although I discuss the topic of software architecture for IDEs at the end of this thesis, my work is mainly concerned with the challenges of improving IDE interaction for touch-based devices. The capabilities of modern devices for *multi-touch interaction* and their support for *gestural interaction* styles enable new tools for touch-centric manipulation and creation of source code. As previously mentioned, textual representation of source code has, compared with its alternatives, a number of intrinsic advantages. Since current mainstream programming languages are optimized for editing in regular text editors, this research can build upon extensive infrastructure. Consequently, the scope and approach of my work is examining the interaction with *textual representation* of source code on *touchscreens* (*touchable source code*). More specifically, this work includes research on the aspects of *editing*, *selecting*, and *creating* source code (Figure 2.6); that is, the research is primarily directed towards the code editor.

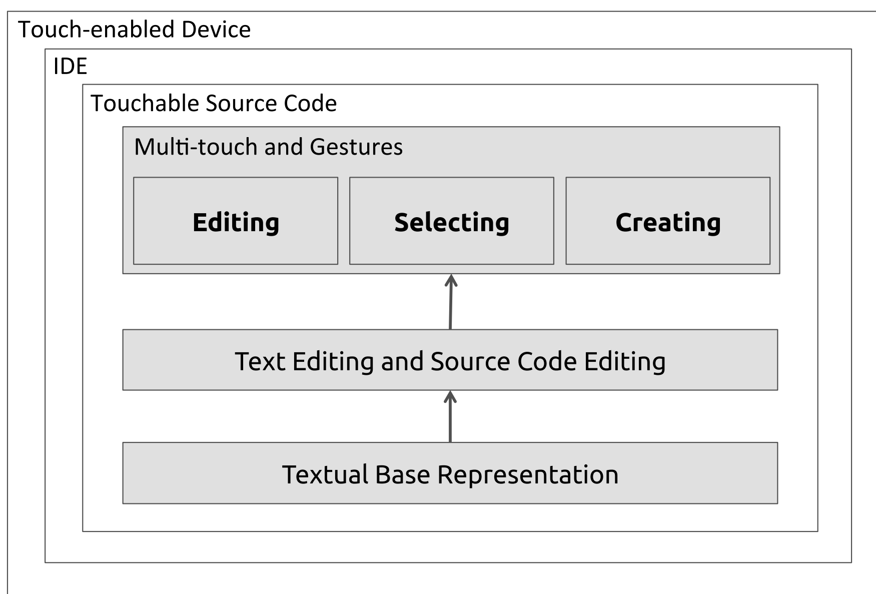


Figure 2.6: Scope and general approach of this work. Devices with touchscreens serve as hardware platform. The IDE natively runs on the target device and supports *touchable source code* consisting of three main layers: Users interact, through multi-touch and gestures, with the text and code editing layer; the code editing layer uses textual representation as basis. The top interaction layer shows the research part of this work (editing, selecting, and creating source code).

Chapter 3

Related Work

In this chapter, I summarize related research on touch-based development environments and interaction methods for code editors. Work presented here focuses on the user interface and interaction, whereas technical aspects and software architecture are separately discussed in Chapter 8. Also, this chapter includes work on improvements to code intelligence features such as code completion, or how researchers have tackled the problem of navigating large code bases. Furthermore, I highlight projects from the field of text entry and editing since textual programming on touchscreens calls for efficient input methods. Finally, I show existing techniques for gesture-driven execution of (menu) commands, and conclude with projects that have applied multi-modal interaction.

My work lies at the intersection of two main research areas: *Human-Computer Interaction* (HCI) and *Software Engineering* (SE). Relevant conferences for referenced work primarily include the following ACM (Association for Computing Machinery) conferences:

- *CHI* (Conference on Human Factors in Computing Systems)
- *UIST* (User Interface Software and Technology)
- *SPLASH* (Systems, Programming, Languages, and Applications: Software for Humanity) with its sister conferences *OOPSLA* (Object-Oriented Programming, Systems, Languages & Applications) and *Onward!*
- *EICS* (Engineering Interactive Computing Systems)

- *MobileHCI* (Conference on Human-Computer Interaction with Mobile Devices and Services)

Relevant work was also presented at:

- *ICSE* (International Conference on Software Engineering) and some of its workshops, such as *CHASE* (Workshop on Cooperative and Human Aspects of Software Engineering)
- *VL/HCC* (IEEE Symposium on Visual Languages and Human-Centric Computing)
- *VISSOFT* (IEEE Workshop on Visualizing Software for Understanding and Analysis) and *SOFTVIS* (ACM Symposium on Software Visualization)

3.1 Touch and Pen Development Environments

Existing development environments for touchscreens could be divided into environments for 1) tabletops, 2) tablets, and 3) smartphones. Classification based on device type or screen size, however, considers only the hardware and ignores the software and its design philosophy. For instance, some environments support structured editing and use graphical programming, whereas other tools allow for flexible input and use textual representation. In this section, I do not strictly categorize related projects based on hardware or software, but briefly present notable work in mostly chronological order.

In Chapter 2, I have briefly discussed GRaIL[1], one of the first IDEs for graphical programming of flow diagrams. Users operated GRaIL with a pen-like input device and a tablet, similarly to how graphic tablets work today. Alan Kay, who had worked with the system at that time, emphasized the advantages of this *direct manipulation* interaction style[2][3]:

> "[...] I used it for half an hour in 1968 and felt like I was sticking my hands right through the display and actually *touching* the information structures directly."

[1] http://www.rand.org/content/dam/rand/pubs/research_memoranda/2005/RM5999.pdf

[2] http://vgable.com/blog/2009/01/18/touching-the-information/

[3] Video source: http://www.youtube.com/watch?v=QQhVQ1UG6aM

Another early work (1972) by Anderson [And72] introduced a special notation for programming on tablets. In contrast to GRaIL, users not only drew symbols such as boxes, but also used textual statements to specify the logic of a program module. Braces and box-like symbols drawn near the right margin of textual statements denoted control statements and conditions (e.g., *if-then*, *do-while*). Anderson pointed out that this notation was more compact, printable, and allowed for easier editing than pure flowcharting. In view of tablet usage, he notes:

> "[...] with hardware costs continuing to decline and with increasing experience in developing flexible character recognizers for use in conjunction with these tablets, the tablet is becoming increasingly viable as a means of communicating with a computer"

While graphic tablets are more widely used today, their primary field of application is digital design and drawing. Software development tools have yet to be rediscovered as potential environment for pen-based programming.

Frisch et al. [FHD09] have investigated diagram editing on *tabletops*, that is, large interactive surfaces where users sit or stand around a table and work on the touchscreen from above. In their study, the authors asked users to interact with the diagram by performing those gestures that they regarded most appropriate for an action. In other words, actual end users rather than experts should design the gesture set. An often cited work by Wobbrock et al. [WMW09] has inspired a number of such studies that elicited *user-defined* gesture sets. After calculating the level of agreement among the participants' gestures, researchers usually present a final gesture set. Designers of similar applications could then use the set as template or recommendation when creating their own applications. Unlike user-defined gestures, however, gestures created by interactions designers or domain experts might result in fewer ambiguities.

Zeleznik et al. [ZBAK10] have demonstrated that multi-touch technology is well-suited for technical work and problem-solving domains. Their system *Hands-On Maths* implements a number of interaction techniques for solving mathematical problems. Users manage virtual documents, write equations holding a pen device, and perform transformations to these equations using multi-touch gestures or bi-manual interaction techniques. Zeleznik et al. have argued that users would better learn and perform their tasks more efficiently if direct multi-touch manipulation and free-form pen input was enabled. Positive feedback from their prototype evaluation has shown that users find working in such environments comfortable and natural. Matulic and Norrie [MN13]

took a comparable approach and combined pen input and gestures for document editing on a tabletop. The system follows the same pattern of replacing certain UI widgets with gestural interaction.

CodePad by Parnin et al. [PGR10] is an environment for maintaining concentration during programming and software development tasks. The project attaches importance not only to the efficiency or naturalness of interaction but also to task coordination so that programmers keep their focus. For this purpose, *CodePad* provides interactive spaces for programming-related tasks on secondary touch-enabled devices. These devices connect to the main IDE and bring additional comfort to development scenarios such as refactoring, visualization, or navigation. The project draws its design ideas from the fact that developers usually deal with different artifacts and vary in their personal working styles. Rather than predicting and displaying relevant information directly in the IDE, a physically separate space enables programmers to interactively manage related content and synchronize with the central system. Parnin et al. call this a “mental playground” for developers and an “additional place to keep their thoughts”. Although the authors have implemented some of their presented features, their work remains a vision and suggests research challenges for future work.

The project *CoffeeTable* [HBKW11], while also supporting multiple devices, aims to enhance the software development process through a central tabletop application. Developers place their laptops onto the surface and then exchange information, visualize the software architecture, or assign elements to their personal laptop workspaces.

In addition to utilizing tabletops, researchers have explored the idea of programming on tablets and smartphones. McDirmid [McD11] has introduced the programming language *YinYang* and discussed its suitability for programming games on tablet hardware. The language is designed around composable tiles with attached identity and behavior. By simplifying the display of programs through this model, *YinYang* trades flexibility for ease of code input. The development environment eliminates cursor movement and selection by tapping on tiles and selection of menu entries—actions that are easier to perform on tablets than code entry via a software keyboard. While the author did not conduct a formal study on input efficiency, his personal test showed that touch input was about 66% slower than regular keyboard input.

Hesenius et al. [HOMH12] pursued a similar strategy for their tablet-based development environment *Touching Factor*. To increase developer productivity, they used

concatenative programming[4] with its more concise syntax. Despite their advantages regarding input efficiency, neither *YinYang* nor *Touching Factor* allows users to program in popular textual languages; also, both projects have not formally evaluated their approach.

The idea to use a smartphone for programming has motivated successful projects such as *TouchDevelop* by Tillmann et al. [TMdHF11]. Since smartphones have become ubiquitous, mobile development environments have the potential to make programming available not only to programmers in developed countries but also to people in developing countries, where a phone may be the user's only personal computing device [TMdHF11]. *TouchDevelop* primarily aims at giving students and hobbyists the opportunity to create applications using solely the touchscreen and the capabilities of their device. Users create their programs in a structured programming language; this structured language, in turn, enables them to edit their programs in an (although not strictly) structured editor that provides primitives for common tasks. The resulting programs run on the devices themselves. Download statistics for the *TouchDevelop* application indicate that its concept has been found valuable by non-professionals.

Ihantola et al. [IHK13] have created a mobile programming environment that is geared towards teaching programming. Their system facilitates slightly more flexible input by extending a block-based approach with selectable options that can change the behavior of individual code fragments. In addition to programming on phones, Nguyen et al. [NCT13] have developed *GROPG*, an application that provides interactive debugging features directly on smartphones, similarly to the functionality of desktop debuggers. Furthermore, projects such as *ScratchJr*[5] or *Hopscotch*[6] have used the iPad as platform to let young children create interactive programs. Both applications mainly employ drag-and-drop interaction with blocks of code for visual construction of programs.

So far, most of the mentioned projects have used variants of structured input to tackle the issue of efficient code input on touchscreens. Biegel et al. [BHLD14] have recently pursued a different approach. Instead of creating new applications from scratch, they have used the popular Eclipse IDE as basis and "touchified" it; that is, they modified the IDE in particular ways in order to optimize its UI for touch input. For example, the authors have changed the arrangement of panels, increased the size of user interface

[4] http://en.wikipedia.org/wiki/Concatenative_programming_language
[5] http://www.scratchjr.org/
[6] http://www.gethopscotch.com/

elements, and replaced menus with touch-optimized alternatives. On the one hand, this approach successfully reduces the technical hurdles associated with porting IDE functionality to touchscreens. Rather than rewriting the entire IDE–a task that would be immensely time-consuming and non-trivial in the case of Eclipse–UI adjustments render the application usable on touch displays. On the other hand, this solution may not be able to take full advantage of natural interaction. As the evolution of operating systems (e.g., Microsoft Windows) towards touch-interaction has shown, new interaction techniques remain limited in expression when the core of the system was originally designed according to the WIMP paradigm. With regard to mobile devices, a touch-optimized application layer cannot simply be modified to match the look-and-feel of the platform, thereby risking to deteriorate the user experience. Therefore, the approach might be most beneficial in situations where the regular desktop version and the "touchified" version are interchangeably used (e.g., on laptops that can be converted to tablets). Techniques, such as those proposed in this work, could additionally be integrated to enhance the editor of the IDE.

3.2 Text Editing

Finseth, author of the exhaustive treatise "The Craft of Text Editing" [Fin91], loosely defines text editing as follows: "In its most general form, text editing is the process of taking some input, changing it, and producing some output". The tool for text editing, the text editor, is operated by humans and since humans have different levels of experience and different goals, the design of text editors "must incorporate knowledge of what task or tasks the user is trying to accomplish" [Fin91]. Finseth lists five basic types of users with varying amounts of experience in computer usage and programming: Neophyte, Novice, Basic, Power, and Programmer-Level. Although "Basic" users, according to this categorization, already understand simple programming concepts, this work mainly targets the last two categories.

Text input and text editing on touchscreens have notoriously been difficult and led to a considerable amount of research on interaction techniques, not least since text operations are crucial for a large number of applications (and for programming in particular). Before I present solutions to text input on touchscreens, I first discuss text editing as practiced in desktop environments since it lays the foundations for mobile

and touch-based solutions. Moreover, I differentiate between work on text *entry* (or text *input*) and text *editing*.

3.2.1 Text Editing in Desktop Environments

Larry Tesler, who contributed the ubiquitous text editing pattern *Cut/Copy-Paste* more than 40 years ago, has stressed the importance of *modeless* text editing [Tes12]. Lesler defines a *mode* as:

> "a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input."

Tesler gives a concrete example of using modes for text editing: In *NLS* (onLine System), an early system by Douglas Engelbart for designing technical specifications, typing the letter "M" activated "Move" mode. After marking the start, end, and destination of the source, invoking an "ok" action completed the command. By introducing modeless *suffix* command syntax, where users specify actions *after* the objects they operate on, Tesler and colleagues could improve error-recovery by using features such as re-selection and *undo*.

On the one hand, modes can have adverse effects on error rates and consequently degrade usability during text editing [Tes12]. On the other hand, text editors such as VIM[7] exclusively rely on modes and still enjoy high popularity with developers. However, VIM is widely perceived as having a steep learning curve for becoming efficient. Furthermore, Tesler argues that modeless editing typically requires fewer keystrokes and button presses, and ultimately saves time. His modeless system *Gypsy* implemented basic text editing features that later became standard: Among other things, *Gypsy* introduced clicking between characters to set the cursor insertion point, dragging down and up to select text, double-clicking to select a word, cut/copy and paste, and searching text via editable input fields.

Another early work on text editing includes Pike's text editor *sam* [Pik87] and *Acme* [Pik94]. In *sam*, users could perform repetitive editing tasks by typing command syntax containing regular expressions. *Acme* aimed at supporting programmers in an editing environment that promoted mouse usage as interaction style for text operations.

[7] `http://www.vim.org`

In contrast to text editors, *structure editors* enforce a *syntax-directed* editing style; that is, users directly modify the syntax tree. For instance, a structure editor for XML documents can prevent users from invalid operations because all edits are checked against a formal schema. Despite their advantages in preventing errors, early syntax-directed editors could not gain wide acceptance due to usability issues [KU93]. The top-down editing style can render individual modifications to source code time-consuming [KAM05b]. To propose improvements to structure editors, Ko et al. [KAM05b] have studied programmers' text editing strategies. In the authors' user study on how programmers approach code editing, they have identified a number of patterns. For instance, editing *names* in declarations and references accounted for 43% of all edits, while editing *lists* accounted for 23% of their data. 15% of all edits were applied to *infix* expressions, 6% to *keyword* structures, 8% to *literals*, and 3% to *comments*. The details of these results indicate the situations where structured editing features might be beneficial and where more flexible, unstructured editing should be supported.

3.2.2 Text Entry on Touchscreens

Text entry on touchscreens has motivated numerous publications on ways of entering characters most efficiently without the presence of a physical keyboard. (Due to the large body of available work, this section only lists selected publications.) Most owners of tablets and smartphones have presumably experienced inaccuracies when using on-screen keyboards or have missed the tactile feedback of physical keyboards. Given the importance of text input, it appears surprising that smartphone manufacturers have only slowly improved the efficiency of their text entry systems. In a study by Findlater et al. [FWW11], typing speed on a flat surface (under ideal conditions) was 31% slower than typing on a physical keyboard.

Zhai et al. [ZKG+09] converted their research project *ShapeWriter* into a (formerly available) Android application. In *ShapeWriter*, users draw a stroke over the letters of a word while the system compensates for inaccuracies or missing letters; the user's intended word is found by interpreting the stroke shape. Later, Bi and colleagues [BCO+12] have extended this concept of drawing over letters to bi-manual input and multiple strokes. *Swype*[8] builds upon similar mechanics as *ShapeWriter* and comes pre-installed on some Android devices. Lately, Apple has introduced *QuickType*[9] in

[8] `http://www.swype.com/`
[9] `https://www.apple.com/de/ios/ios8/quicktype/`

iOS 8. *QuickType* improves typing by predicting appropriate words and phrases based on the user's context (e.g., whom the user writes to or which application he uses for writing). Apple has only recently opened up their APIs for integrating system-wide custom keyboards.

Furthermore, there has been a number of publications on performance characteristics of different keyboard layouts [LGYT11], adapting keyboard layouts to the user's grasp [CLWC13], and on algorithmic optimizations for the word correction and completion features of text input systems [BOZ14]. Researchers have employed strategies such as zooming to enlarge the small touchable areas of keys [PWM14, OHOW13] or used the back of the device (instead of the front) as interaction area [SO13]. Findlater et al. [FLW12] have proposed combining bi-manual interaction and multi-touch gestures for entering alternative characters and punctuation. Moreover, researchers have explored ways of gesturing over ordinary keyboards [ZL14], and performing gestures while hovering over the keyboard surface [TKH+14].

Kristensson et al. [KBC+13] have noted that, despite progress in the field of text entry methods, "the research community is scattered across different fields, such as human-computer interaction (HCI), natural language processing (NLP), speech processing, and augmentative and alternative communication (AAC)". They have listed three "grand challenges" in text entry, namely: 1) generally improving performance, in particular on mobile devices; 2) better supporting the diversity of writing systems and languages; and 3) providing appropriate methods for users with disabilities.

3.2.3 Text Editing Gestures

Editing text includes operations such as selecting words or paragraphs, deleting lines, or placing the cursor between characters. The advent of touch-enabled devices has stimulated research on how traditional text editing features can be adapted to take advantage of *gestures*. However, researchers considered gesture-driven text editing earlier than one might expect. In 1987, for example, Wolf et al. [WMS87] explored the use of hand-drawn gestures for text editing. (Wolf et al. note that the earliest work on gestural text editing dates back to 1969, but those projects focused more on design and implementation aspects.) The authors have listed four advantages of this interaction style. Gestures 1) allow users to specify the command *and* its arguments in a single action; 2) improve learning and recall due to their spatial form; 3) enhance the user's

sense of directly manipulating objects; 4) mimic the user's accustomed working styles when using pencil and paper.

Furthermore, early work by Goldberg and Goodisman [GG91] from Xerox PARC (Palo Alto Research Center) describes how the researchers explored the use of a stylus for text manipulation. Although some of their efforts focused on the technical issues of gesture recognition–which is today more reliable due to improved algorithms–they also highlight design principles. For example, they stress the importance of *not* attempting to imitate paper, but rather focusing on tasks where stylus systems are superior in comparison with "analogue" interaction style.

More recently, Fuccella et al. [FIM13] have investigated gestural techniques for editing text on touch-devices such as smartphones. The authors have remarked that research on text *editing* has not received the same amount of attention as research on text *entry*. Their approach incorporates drawing gestures for caret movement and text selection on top of the manufacturers' on-screen keyboards. This gestural layer is optional and may or may not be used depending on personal preference. In their user study, participants who had used the gestures could increase performance between 13% and 24%. The technique of Fuccella et al., however, suffers from the disadvantage of requiring the keyboard–which covers a considerable portion of the screen–to be visible while performing the gestures.

In another recent work, Leiva et al. [LAV13] have proposed *MinGestures*, simple directional gestures for text operations like deleting, inserting, merging, or splitting. Their research is driven by disambiguating text editing operations from handwritten text, and the results primarily apply to post-editing interfaces.

Text editing systems as provided by the operating systems of smartphone manufacturers typically display UI widgets for text manipulation. Users perform editing tasks through a combination of dragging selection handles and triggering actions in pop-up menus. Smartphone operating systems have not yet included gestural text editing capabilities by default, but users can take advantage of such features by installing a number of third-party applications from "App Stores".

In summary, there has been some progress in gestural text editing, but currently only few users take advantage of editing gestures, and there is no agreed upon gesture set for standard operations.

3.3 Editor User Interfaces and Interaction

In this section, I highlight novel approaches in user interface design for development environments. Additionally, I present specific techniques for intelligent source code editing in code editors and show how researchers have tackled the issue of efficiently navigating large code bases.

3.3.1 Novel Editor Interfaces

(*Note:* The following two paragraphs on the projects *Code Bubbles*, *Code Canvas*, and *Debugger Canvas* are a slightly reworded version of my related work section in [RWE13]). A number of research projects on novel editor UI concepts could be labeled as *canvas-based editing*. *Code Bubbles* [BRZ+10] has been much-discussed in the programming community and has continued development after its first presentation. *Code Bubbles* seeks to improve code understanding and maintenance; it disregards the file-oriented nature of existing IDEs and instead shows code fragments that appear as connected, interactive *bubbles* on a pannable 2D canvas. Instead of constantly switching views, the tool automatically groups these concurrently visible bubbles into working sets for the task at hand. Simultaneous code views simplify code inspection since they visualize calling sequences and assist developers in understanding the program flow. The developers of *Code Bubbles* have demonstrated that their metaphor significantly reduces the time spent navigating and the time needed to complete code understanding tasks.

A similar project, *Code Canvas* [DR10], also takes advantage of spatial memory in order to reduce disorientation. Using a canvas, linked code fragments, semantic zoom, and information overlays, it serves as an interactive map for developers. Since *CodeBubbles* and *Code Canvas* have shared some ideas, a collaboration between both projects has finally led to the industrial tool *Debugger Canvas* [DBR+12]. The tool is now part of the debugging facilities of Microsoft Visual Studio. In *Debugger Canvas*, a map-like, zoomable surface supports debugging tasks by displaying call paths and execution traces in a set of connected code fragments (bubbles). Developers can then step back and forth through the code and visually explore relationships. This feature is particularly helpful when working with unfamiliar code bases and might help programmers forming a better “mental model” of the program. However, the authors of *Debugger Canvas* acknowledge that their representation may not be as beneficial

when developers are familiar with a code base or when they work on smaller problems. (The tool is realized as separate mode within the main IDE so that switching incurs a certain overhead.) Since the previously mentioned projects use a zoomable canvas and do not exclusively rely on traditional user interface elements, they might work well on touchscreens after adding support for multi-touch interaction.

Other researchers have proposed solutions to augment editors rather than replace the entire UI. For instance, French et al. [FKD13] have integrated visual programming elements into textual source code. Since the authors acknowledge that visual programming suffers from the frequently cited *scaling-up problem* [BBB+95], they display visual elements only when appropriate (e.g., for images, editing tables, or visually editing regular expressions). This is in line with other work on *hybrid environments* such as *Barista* [KM06], which I have already mentioned in Chapter 2.

Furthermore, I have mentioned *live programming* approaches in Chapter 2. Likewise, these projects fall under the category of "novel editor interfaces" since they provide additional UI elements for interactive and real-time manipulation of an executing program. Apple's recently introduced *Swift* programming language and a feature called *Playgrounds* enable such functionality in the Xcode IDE. This integration demonstrates that mainstream languages have now started integrating promising concepts that were already used decades ago [HW85] and officially introduced as *live programming* in Hancock's dissertation (2003) [Han03][10].

3.3.2 Intelligent Code Editing

The term "Intelligent Code Editing" refers to IDE features that provide programmers with help, hints, and recommendations during code editing. Most major IDEs inherently support code intelligence features for a number of programming languages, whereas general-purpose text editors often require add-ons. Code intelligence is usually realized by means of parsing syntax trees, analyzing usage patterns, or even taking advantage of collective intelligence through crowd-sourcing techniques. After analyzing the code, the IDE typically displays UI widgets or lets programmers fine-tune results by selecting menu entries. The ultimate goal of code intelligence is speeding up programming and preventing users from programming errors.

[10] `http://lambda-the-ultimate.org/node/4715`

Code Completion

Code completion (or *auto-complete*) could be regarded as one of the most basic and–due to growing APIs and the proliferation of code libraries–principal forms of code intelligence. Code completion encourages developers to explore APIs without having to switch to separate windows for documentation. The mechanics of code completion usually work similar to the following pattern: After each entered keystroke, the IDE refines a list of displayed suggestions; the programmer selects a suggestion from a menu and *completes* part of the code by inserting the suggestion through a keystroke or mouse click. (In Eclipse, auto-completion is called *content assist*). Since the list of matches can be large and may contain irrelevant entries, most scientific work on code completion suggests ways of improving the quality and order of the displayed matches. For example, improvements beyond analyzing the program structure have been based on program history [RL08, LHKM13], or crowd-sourcing channels such as mined code repositories [BMM09, ZYZ$^+$12] and code examples [MFSM10]. For a recent comparison of different techniques, see [ARSH14].

Omar et al. [OYLM12] have extended code completion with an approach called "Active Code Completion". While IDEs like Eclipse display only menu entries and static documentation next to suggestions, *Active Code Completion* shows *palettes*; that is, custom widgets for code generation. For instance, when a command involving colors is about to be completed, a *color palette* renders different color values and a search field for color names. Another example presented by Omar et al. is a widget for testing regular expressions before they are inserted into the code. Furthermore, the authors have developed general guidelines for their concept, which could prove to be useful when designing similar IDE features.

Errors and Visualizations

Since programmers rarely write code free from errors, features such as highlighting syntax errors, pointing out bad coding practices, or suggesting fixes for potentially flawed code, have become standard in modern IDEs. Eclipse, for example, displays small light bubbles called "Quick Fixes" in the editor gutter when the static analysis engine detects errors. After a keystroke or click on the bubble, developers can then select an option to let the IDE automatically fix the code. However, despite their benefits, research has repeatedly shown that developers tend to ignore results of

static analysis tools due to false positives, large volumes of generated warnings, and inappropriate presentation [JSMHB13]. This is why other projects have attempted to improve visualizations of problematic code while avoiding information overload and distractions [MHB10, LvdH11]. The issues commonly associated with error reporting and debugging have led to tools that allow developers to ask the IDE questions on *why* errors have occurred, and to tools that display real-time information about the run-time behavior of programs in the gutter area of editors [LBM14].

In addition to improving error visualization, new approaches have exploited crowd-sourcing techniques to help programmers with interpreting error messages and recovering from errors. *HelpMeOut* [HMBK10], for example, assists users in fixing errors based on other developers' past solutions. The tool presents fixes in a separate panel that explains the error message, allows voting solutions up and down, and lets programmers automatically integrate the suggested fix into their source code. Fast et al. [FSW+14] have taken the idea a step further and integrated *emergent behaviors* into the IDE; that is, tools driven by knowledge about how developers *actually* use programming languages. This knowledge-base fosters applications for uncovering bugs that occur when programmers differ from common idioms and language conventions (i.e., when code is unlikely to appear in practice).

Example-centric Editing and Help

Bruch et al. [BBMM10] have previously labeled IDEs that integrate the knowledge of masses as "IDE 2.0"–following "Web 2.0", a term that was introduced for crowd-based web applications about ten years ago. Researchers have devised a number of projects that apply crowd-sourcing methods for automatic recommendation of code examples and code snippets. Conventional IDEs support code snippets (short pre-defined blocks of code) either through auto-completion features or user-defined *code templates*. Programmers can trigger code templates via keyboard shortcuts and then adjust template placeholders to fit their needs. Integration of more complex code examples, on the other hand, first involves manual copying from documentation or web sources, and then adapting the examples to the current context.

Programming by Example (or *Example-centric Programming*) is the research area that has dealt with advanced strategies for integrating code examples into the programming workflow. The projects *Codelets* [OB12] and *SnipMatch* [WYBV12] appear particularly interesting from a UI perspective.

Codelets treats code examples as searchable "first-class" objects and displays an interactive adjustment interface as embedded element. This helper-widget allows the programmer to configure the code based on criteria that documentation authors have defined in advance. In addition, the widget draws connection lines to parts of the surrounding code and stays synchronized after edits. In a user study, users of *Codelets* could complete example-based tasks 43% faster. Although the authors of *Codelets* claim that authoring such examples is straightforward, their concept might suffer from a lack of prepared third-party *Codelets* for the multitude of available languages, frameworks, and libraries.

SnipMatch supports example-centric programming by taking the local code context (e.g., variable names, cursor position, dependencies) into account when searching and ranking snippets. After triggering the search interface via a keyboard shortcut, users can scroll through matches and preview the snippet inline as it would appear if the code context was considered. Following the selection of a snippet, programmers can change arguments in a dialog box that reflects all edits in the code while the user is typing. Authors must prepare the snippets, but due to the similarity to editing code templates in IDEs, the process seems more lightweight than with *Codelets*. The central idea is that shared repositories grow over time as users contribute more snippets.

3.3.3 Code Navigation and Search

Besides editing source code, navigating *between* source files and *within* source files takes a considerable amount of a developer's time. In a study by Ko et al. [KMCA06], developers spent 35% of their task time on navigating source code. Developers spend more time reading than writing code, for example, when diagnosing errors or attempting to understand unfamiliar code bases. As a result, navigation facilities of IDEs for fast moving between different parts of a program, and for quickly locating specific code, are crucial for productivity. Ko et al. also noted, "Eclipse's navigation tools caused significant overhead [...] by opening new tabs and requiring a return navigation". Programmers can quickly get lost in large programs, but most IDEs do not make these digressions visible to the developer. After having examined Eclipse usage data of 67,500 Java developers, researchers discovered that only 6% of developers had used call graph tools for navigating method chains, and only 18% had used tools for navigating the inheritance hierarchies of an object-oriented program [VM10]. These

findings were attributed to poor usability and lack of discoverability of the built-in navigational aids of IDEs.

Mylyn[11], a popular Eclipse extension, attempts to improve navigation by creating a "task-context" that automatically collects and displays relevant artifacts. This automatism, however, might also lead to incorrect assumptions about the developers' intentions. *NavTracks* [SES05], another frequently cited work, enhances browsing by recommending files that programmers might need at hand, although this approach could be criticized for similar reasons as *Mylyn*. To support the navigation of software artifacts, the previously mentioned canvas-based editors such as *Code Bubbles* [BRZ+10] and *Code Canvas* [DR10] exploit the users' spatial skills. However, Parnin et al. [PGR10] cast doubt on the effectiveness of previous approaches by arguing, "predictive guesses are rarely correct [...] and spatial visualizations break down when developers must transition to spatially distant locations [...] increasing the likelihood to become disoriented when panning and zooming." He proposes displaying additional information in form of *waypoints* on top of code and *navigation trails* between waypoints. Henley and Fleming [HF14] have expressed similar concerns about canvas-based editors leading to "on-screen clutter", navigation errors, and the increased time spent with having to rearrange code fragments.

Krämer et al. [KKK+13] have stressed the importance of promoting call-graph navigation. They have compared the tools *Blaze* [KKKB12] and *Stacksplorer* [KKD+11] to traditional call-graph tools found in IDEs. *Stacksplorer* shows automatically updated method callers and callees in columns next to a focused method in the editor, whereas *Blaze* displays a single path through the currently focused method. Both tools could decrease completion times in user studies involving maintenance tasks. Krämer et al. note that developers employ less effective strategies, such as text searches, when IDEs do *not* provide call-graph tools.

Improving on previous work, Henley and Fleming [HF14] have designed *Patchworks*. This code editor displays a grid of 3 x 2 fixed *patches* with each patch holding a code fragment. Users can move the grid to the left and right and use a *ribbon view* as overview or for adjustments of the patches. Because the grid is restricted to one dimension, users tend to be less susceptible to navigation mistakes than on 2D canvases. The results from a user study comparing *Patchworks* to *Code Bubbles* and Eclipse showed

[11] `https://www.eclipse.org/mylyn/`

that *Patchworks* users made fewer navigation mistakes, navigated faster, and overall reacted positively to this concept.

3.4 Commands, Menus, and Gestures

The proliferation of functionality in IDEs presents challenges for efficiently invoking commands. Command execution in desktop IDEs has been driven by classic WIMP interaction style, such as extended context menus and an abundance of keyboard shortcuts. Due to the absence of a mouse and keyboard, touchscreens require alternative methods comprising multi-touch and gesture-driven controls. Such interfaces are often referred to as *NUIs* (Natural User Interfaces); however, the sole use of touch-based interaction does not necessarily imply that an interface feels "natural" to the user. Norman [Nor10], for instance, referred to NUIs as "marketing name" and stated:

> "The strength of the graphical user interface (GUI) has little to do with its use of graphics: It has to do with the ease of remembering actions, both in what actions are possible and how to invoke them."

His essay continues by pointing out the disadvantages of gesture-driven systems such as lack of visibility, feedback, and discoverability. According to Norman, however, these drawbacks can be overcome when following "basic rules of interaction design". Although Apple's iPhone has clearly had tremendous influence on popularizing NUIs, in their *Human Interface Guidelines*[12], for example, they recommend cautious use of well-established standard gestures (e.g., the well-known *swipe* and *pinch* gestures); complex gestures should be avoided as the only way of performing actions.

HCI research on multi-touch interfaces has had a long history[13]. Researchers have sought ways to make gestural interaction more efficient and to address the above-mentioned concerns. In this section, I concentrate on presenting methods that are most relevant to the IDE introduced in later chapters. Proficient users of code editors gain much of their efficiency from keyboard shortcuts; touch-optimized methods for triggering commands are therefore of particular concern here. Due to advantages regarding cognitive factors, the use of gestural *strokes* as command shortcuts [AZ09] or

[12] `https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/`

[13] `http://www.billbuxton.com/multitouchOverview.html`

other stroke-based menu designs have often been suggested as activation mechanism. An argument in favor of this approach is motivated by the fact that keyboard shortcuts can be hard to remember because often no intrinsic mapping between a command and its shortcut exists.

One notable method exploiting strokes has been designed by Kurtenbach [Kur93]. He has introduced *marking menus* and carried out a number of studies on using the technique for efficient command invocation. *Marking menus* are circular menus where selections are performed by stroking (or *marking*) towards an item. Since the menu is displayed after a short delay, expert users who have internalized the item directions can rapidly execute a command without being distracted by the visual representation. According to Kurtenbach, the essential design principles of *marking menus* are “self-revelation, guidance and rehearsal”:

> “Self-revelation means a marking menu reveals to a user what functions or items are available. Guidance means a marking menu guides a user in selecting an item. Rehearsal means that the guidance provided by the marking menu is a rehearsal of making the mark needed to select an item.”

Menus can be nested, which results in “zig-zag” gestures when selecting an item at a deeper level in the hierarchy. Later, *multi-touch marking menus* [LGF10] have been developed as touch-compatible extension. Since this version exploits multi-finger chords for selecting nested menu items, it might be more suitable for two-handed operation or tabletop applications.

Marking menus are an attractive choice for contextual actions because they target both novices and experts. Owing to their simple directional and pre-defined gestures, designers are freed from the burden of making up custom gesture sets. This is further supported by the finding of mark-based gestures being faster and more accurate than free-form gestures [BNLH11]. Because of the elegance of their design and their efficiency, this work makes use of *marking menus* as means of providing gestural access to commands. In Chapter 7, I present the details of this method and an extension that lets users quickly repeat standard code editing actions.

Kurtenbach’s work has inspired numerous variations of *marking menus*. Several studies have sought to extend and improve on the original design but only few attempts appear to be viable in mainstream user interfaces. However, researchers have also developed alternatives with straightforward designs. With *FastTap* [GCS+14], for example, an item of a displayed grid is selected by using one’s thumb and finger (corresponding to

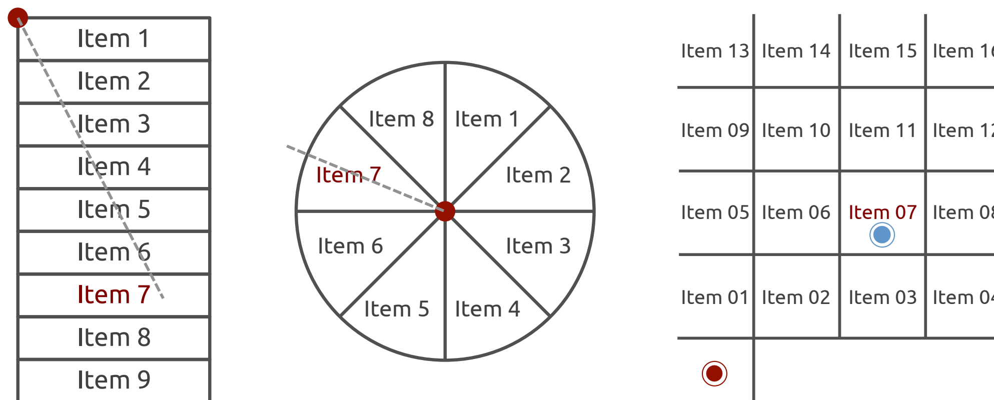


Figure 3.1: Exemplary selection points for a menu item in different menu layouts. *Left:* A regular linear menu. More items are usually revealed when users perform up-and-down *swipe* gestures. *Middle*: A radial menu. In contrast to linear menus, the distance to select an item remains constant since items are put at different angles around the menu center. (Eight slices are preferable). This enables fast directional selection gestures without having to display the menu. Radial menus can be nested, leading to "zig-zag" selection gestures when an item at deeper levels in the hierarchy is selected (*marking menus* [Kur93]). *Right*: A grid-based menu for two-finger selection. The grid is displayed after an activation tap onto the button in the corner. Fast selection without displaying the selection-grid is performed by *chording* with the thumb and forefinger (*FastTap* [FC12]).

an activation tap and selection tap). In expert-mode, chorded thumb-and-finger taps speed up selection by virtue of similar principles as in expert-mode of *marking menus*, that is, through spatial and muscle memory. While *FastTap* might be an appropriate choice for global UI actions due to its fixed activation button, *marking menus* may be easier to integrate as context menus.

Another menu design, called "Under-the-Rock Menus", has been devised by Zeleznik et al. [ZBAK10]. When the user starts dragging an object to a target location, a semi-transparent radial menu "grows" at the starting point. Moving the finger back to the menu and selecting an item allows the user to change the default operation for the dragging movement. In a source code editing context, for instance, such a design could be applied to targets where a dragging action must support multiple different operations (e.g., cut/copy), while the most frequently used operation is set as default. Thus, users could perform certain actions without requiring any menu selection. Figure

3.1 illustrates the different designs of the three mentioned menu layouts (linear, radial, and grid).

Other options for initiating operations include bi-manual techniques such as those suggested for the *BiPad* toolkit [WHM12]. Although these methods enable new interaction techniques, they are not considered for this work since users have reported that two-handed interaction feels unnatural [ZBAK10] and generally preferred one-handed use when possible [ZBAK10, KBCV08, FHD09]. Moreover, supporting bi-manual interaction may complicate the implementation of applications and requires integrating additional instructions for the user.

Apart from the design characteristics of menus, the choice of *which* commands should be visible to the user and how to categorize them is widely regarded important. Dividing commands into categories of related items or alphabetically ordering commands are obvious choices. Findings of the effectiveness of ordering strategies are contradictory, although it has recently been argued that functional groupings should be preferred due to advantages in learning the commands [Sam13].

Furthermore, predictive models may enhance selection by showing users only those commands that they will most likely execute next. In a study by Parnin et al. [PGR10], lists that promoted the four most recently selected documents could cover almost 70% of navigations between documents. A more advanced model than simple recency-based and frequency-based lists has been suggested by Fitchett and Cockburn [FC12]. Their algorithm *AccessRank* considers multiple factors (e.g., temporal information about item access) for determining the most likely items. Additionally, *AccessRank* provides list stability; that is, it minimizes automatic (and potentially confusing) re-ordering of item locations.

As this section has shown, implementors of touch-based UIs can draw on a variety of command execution methods without resorting to keyboard shortcuts. Although keyboard shortcuts could be enabled on a virtual keyboard, the user experience may deteriorate when WIMP interaction style is enforced in NUI environments. Despite the known issues associated with gesture-driven interaction, the advantages of conforming to the inherent paradigm of the target platform should outweigh adverse effects.

3.5 Multi-modal Development Tools

This final section of related work highlights projects incorporating more than a single input channel for interaction. So-called *multi-modal* applications combine information from multiple input sources such as speech, gaze, or pen input. Researchers have argued that multi-modal systems are capable of reducing errors, increasing flexibility, and adding more expressiveness to user interfaces [DLO09]. Since these attributes appear to be particularly desirable in complex application areas, this raises the question of how the domain of software development could benefit from multi-modal concepts.

Bolt's system *Put-That-There* [Bol80] has been an early and frequently cited work on multi-modality. Through voice and pointing gestures, a user sitting on a chair could move and modify graphical objects on a large display wall in front of him. The combination of input modalities, according to Bolt, made interacting with the system more expressive. Since then, a large body of work has been devoted to joining input sources in ways that outperform uni-modal interaction or motivate novel UI concepts. For instance, Hinckley et al. [HYP+10] have advocated tools combining touch and pen input for richer interactions. The authors have demonstrated an example where a user keeps holding two fingers onto an object overlapping a photo while his other hand is moving the pen along the edges of the object to cut the traced part out of the photo. The work of Hinckley et al. has proposed a number of such interactions allowing users to perform object selection in tandem with mode switching.

Besides pen and touch interaction, researchers have combined direct manipulation with natural language to make complex editing interfaces more approachable to end users. The project *PixelTone* [LDW+13], for example, runs on a tablet as multi-modal user interface for image manipulation. Voice commands (e.g., "make this greener") and the simultaneous setting of target areas via touch input (e.g., circling an area) enable a rich input vocabulary for photo editing. *PixelTone* also lets users first tag objects through natural language and pointing ("this is Bob") and then refer to these tags later ("make Bob brighter").

As far as software development is concerned, only little work has been done to exploit the potential benefits of multi-modal interaction. *Code Space* [BDHM11] blends touch with in-air pointing to form hybrid interactions that support developer meetings. The system allows a team of developers to discuss code or distribute tasks using

personal smartphones and a shared multi-touch wall. In-air gesturing is employed as a means of manipulating on-screen code fragments, annotating code, or transferring data. The previously mentioned project *Hands-On Maths* [ZBAK10] has utilized hybrid multi-touch and pen-based interaction for algebraic transformations and document management in a virtual-paper-environment.

To explore multi-modality in a code editing context, I assisted in supervising the design and development of *EyeDE* [GREW14]. *EyeDE* is the prototype of a code editor that contributes eye-tracking as input modality in addition to regular mouse and keyboard usage. The editor runs in a web browser and processes gaze data received from an eye-tracking server. Since the user's gaze point and focus of attention often match, interface elements can intelligently adjust in the current field of view without involving substantial physical effort from the user. For instance, *EyeDE* automatically provides navigation aids when the user keeps gazing on an activation trigger displayed above methods calls (see Figure 3.2); that way, the programmer could jump to the declaration of a method. Other examples include displaying the body of a method below method calls, highlighting all occurrences of a variable, or looking up documentation. In addition, the system lets users switch files and scroll within files through gaze control.

Using eye-tracking information in a code editing context seems promising: Development environments could reduce the cognitive effort and increase the user experience by inferring which information and tools programmers need at hand. However, gaze-based interaction has not been without its issues. First, the inherent inaccuracies of eye-tracking impede precise calculations at the character-level in source code. Second, since the eyes are "always on", user interfaces require methods for resolving the issue of unintended command triggering (the so-called *Midas Touch problem*). Finally, advanced tracking technology is not yet widely available in consumer devices.

Opportunities for extending *EyeDE* may include the integration of natural language, similar to *PixelTone*. For instance, after selecting a block of code via gaze controls, users could apply code transformations via voice commands (e.g., "extract this into a new function"). Begel and Graham [BG06] have designed a spoken version of Java and concluded that programming by voice, although feasible, is slower, and programmers are averse to dictating code. Since the keyboard is usually the most efficient and accurate way of editing textual code, adding voice support may thus be most beneficial for handicapped users or in particular environments.

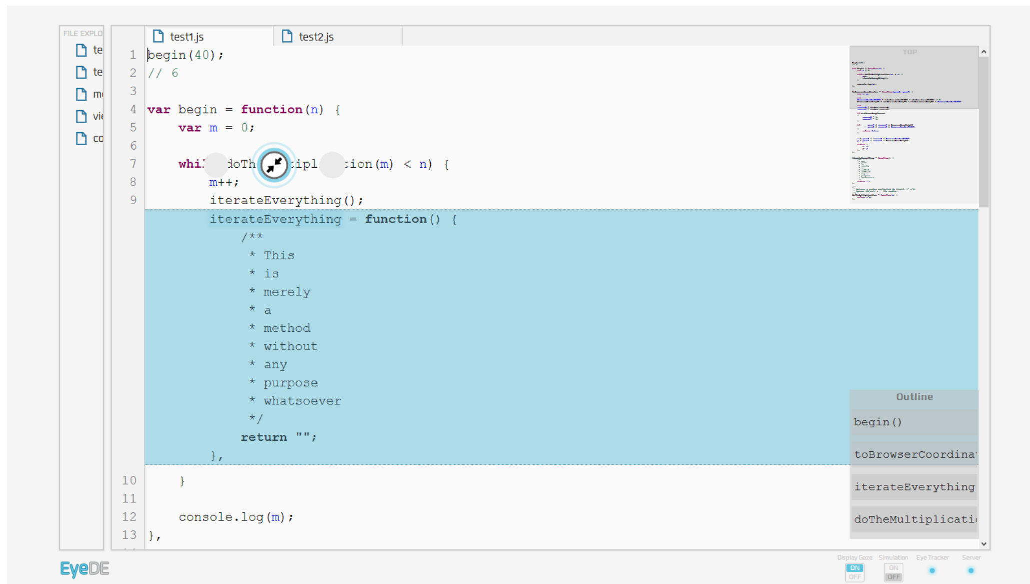


Figure 3.2: *EyeDE*, a gaze-enhanced IDE prototype developed by Glücker et al. [GREW14]. The screenshot shows the expanded body of a method after the user had gazed on the trigger ("bubble") displayed above the method call.

Although my work does not explicitly employ multi-modal techniques, work presented in Chapter 4 includes a study on pen usage for source code editing. It remains future work to explore how multi-modal interaction techniques could further enhance software development environments.

Part II

Source Code Interaction

Chapter 4

Editing Source Code

Source code editing comprises different types of operations, ranging from simple text editing operations to more complex structural transformations. In order to address the issue of *editing* source code on touchscreens, I conducted a user study. The main purpose of the study was finding out how users would apply gesture-driven interaction for manipulating the textual representation of source code. Furthermore, the study has given insights into the advantages and disadvantages of interacting through a pen device, compared with interaction through conventional touches.

I first introduce the mechanics of typical code editing operations, including behavior-preserving structural transformations. In the second part of this chapter, I report the details and findings of the user study, and discuss implications for the design and implementation of gesture-driven code editors.

The study has previously been published in [RWE13]. This chapter is an exhaustively revised and extended version of the original paper.

4.1 Introduction

According to Pressman [Pre10], software organizations often spend up to 70% of their resources on maintenance work. Maintenance-oriented activities in IDEs include reading and understanding the code base, navigating dependencies, searching relevant code, editing code, and testing [KAM05a].

Before actual code editing, developers first need to understand the program. Ko et al. [KMCA06] have observed that developers pass through three main stages when they attempt to understand unfamiliar source code: They first explore the code by *searching* until they find certain cues; when the search process reveals valuable code fragments, they start *relating* the code by navigating dependencies forward and backward; finally, they *collect* found information and start editing as soon as they regard the collected information as sufficient.

4.1.1 Code Editing Operations

Although the act of code editing obviously includes *typing* new code, in this work I differentiate between *editing* code and *creating code*. (The latter is addressed in Chapter 6). Here, the term "editing code" is understood to mean editing and transforming already *existing* code instead of writing new code from scratch. Concerning the type of operations, the process of code editing could be divided into four main categories (Figure 4.1):

Text Editing Operations
: At a basic level, developers edit code by applying standard text editing operations, such as inserting and deleting characters or executing cut/copy-paste commands. In that regard, editing source code is comparable to editing text in a word processing application.

Code Editing Operations
: Developers also perform operations *not* commonly found in word processing applications. For example, code editors provide commands for commenting selected source code in and out or for manipulating lines of code (e.g., swapping, duplicating, splitting, or joining lines). Thus, code editing operations could be regarded as more advanced text editing commands operating on the textual structures found in source code (i.e., tokens, lines or blocks).

Code Intelligence Operations
: IDEs support code intelligence features for automating a series of otherwise manual edits. For example, IDEs highlight programming errors and provide shortcuts for automatically applying a suggested fix (e.g., the *Quick Fix* feature in Eclipse) or implement convenience features for local code transformations (e.g.,

the *Quick Assist* feature in Eclipse). Particular operations (e.g., *Auto-complete* and *Quick Fixes*) not only *edit* existing code but also *create* new code.

Refactoring Operations

Most major IDEs contain commands for applying structural transformations without changing the intended external behavior of the code. These so-called *refactoring* actions are semi-automate mechanisms for increasing the code quality and hence free developers from having to perform multiple steps of a single refactoring manually.

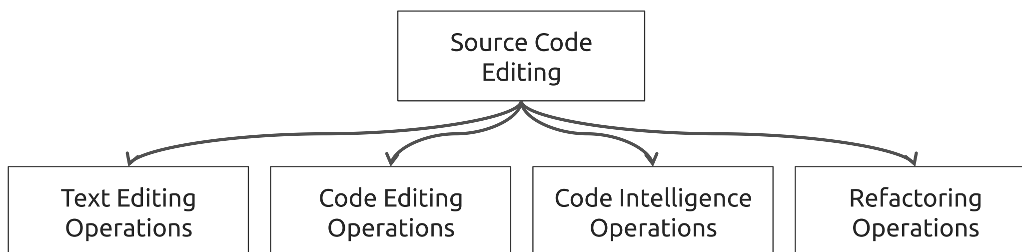


Figure 4.1: Four principal types of operations for source code editing.

Figure 4.2 illustrates these four different types along two axes. The horizontal axis indicates the degree of automatism performed by the IDE, as contrasted with manual execution by the developer. Most automated operations are, in fact, semi-automated since they require user intervention, that is, selecting menu entries or confirming dialog boxes. The vertical axis indicates the level of abstraction at which the operation executes.

For instance, text editing operations such as manually inserting a character at the beginning of a string apply changes to the textual representation (i.e., at the source code level), while refactoring operations automatically apply a series of transformations to the AST (Abstract Syntax Tree). The AST is an internal tree representation of the source code. The developer does not directly interact with the AST but modifies its textual representation. The details of this functional interaction between the AST and the textual representation are language-specific and differently handled by corresponding IDE modules (also see Chapter 8). When further details about the context and dependencies of an edit are required, direct syntactical changes to the source code representation tend to become less useful. Instead, the IDE performs semantic analysis and transforms the AST, which is projected back to the textual representation. Operations involving the AST and file dependencies may result in more

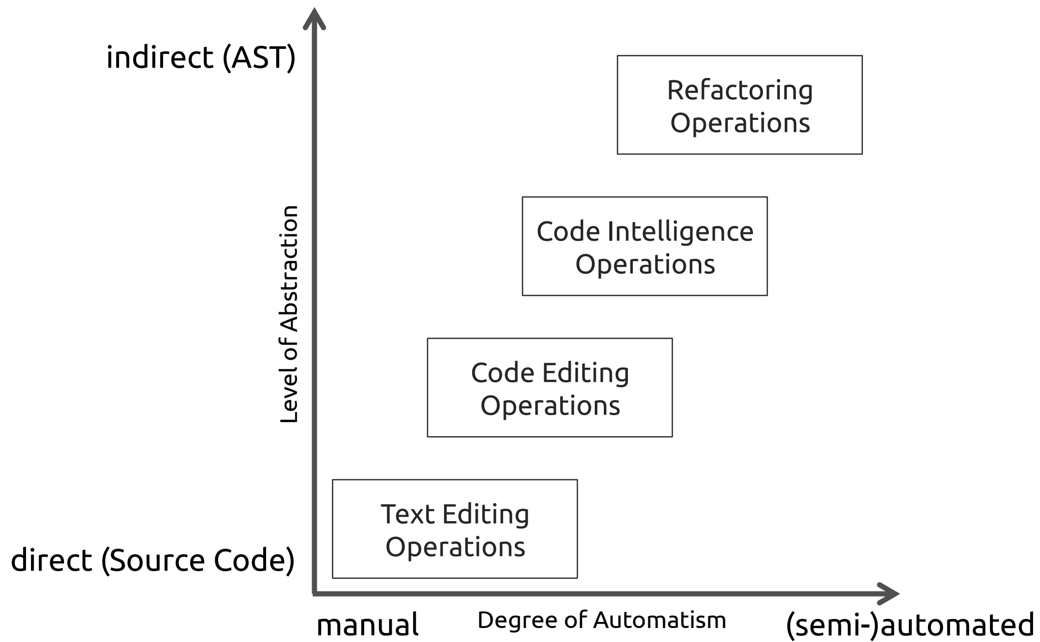


Figure 4.2: Types of code editing operations arranged along two axes. The horizontal axis shows the degree of automatism, ranging from manual developer execution to semi-automated execution by the IDE. The vertical axis shows the level of abstraction, ranging from direct manipulation of the source code to semi-automate transformation of the AST.

fundamental changes of the textual representation. Therefore, the IDE often gathers user input to determine the scope of the change or to resolve ambiguities. In contrast, manual edits of the source code are “cheaper” to perform since they are direct and keyboard-driven. Executing code intelligence and refactoring operations, on the other hand, can be regarded as indirect interaction style since it may entail selecting menus or confirming dialogs.

4.1.2 Code Editing Triggers

Code editing operations are typically triggered via the following UI mechanics:

Keyboard Shortcuts
: Keyboard shortcuts for text editing operations (e.g., for cut/copy-paste) are largely identical across applications, while the mappings for code editing operations tend to be arbitrary and dependent on the IDE.

Application Menu Selection

IDEs usually display editing commands in global *edit* and *refactoring* menus. Some commands show the default keyboard shortcuts next to the item labels. Menu items that cannot be selected are grayed out.

Context Menu Selection

Dependent on the location of invocation, right-clicking on source code displays context-specific operations from all application menus. Disabled items are either grayed out or not shown at all.

Gutter Menu Selection

The gutter area shows icons for errors and suggested fixes. Clicking on an icon presents the available operations in a pop-up menu, requiring further selection via the mouse or keyboard.

Inline Menu Selection

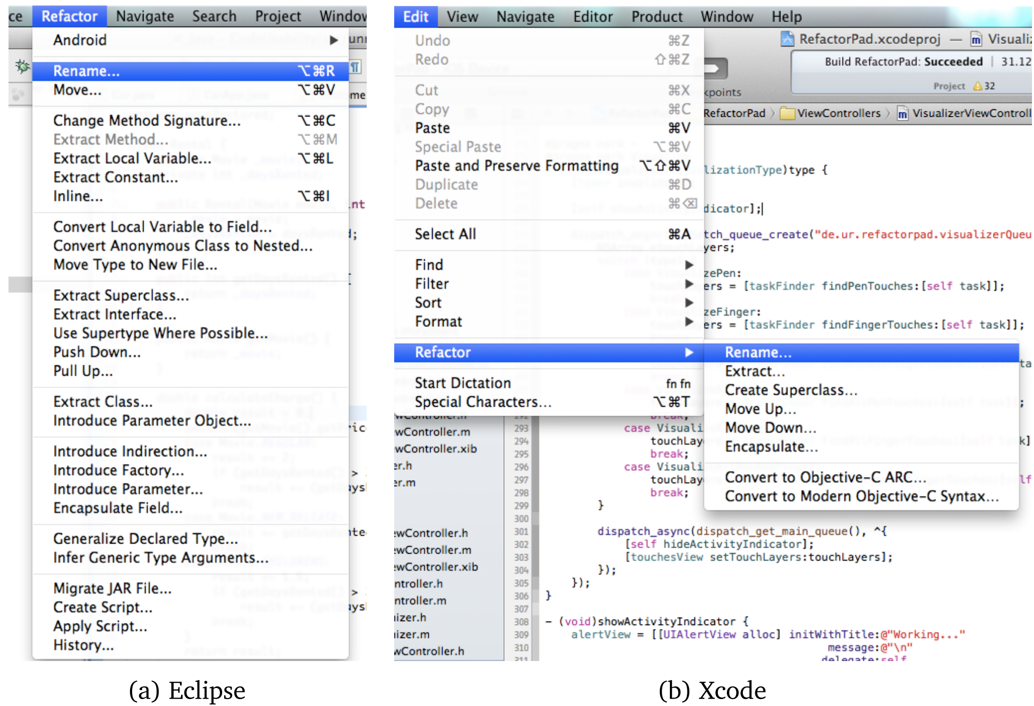
Similar to options in the gutter menu, suggested fixes and local transformations can be triggered by clicking on an icon displayed between source code lines, followed by selecting an entry from the pop-up menu.

Drag-and-Drop

Mouse-based cut/copy-paste operations are initiated by dragging selected code to new locations. The mode of the operation (cut or copy) can be set by holding a modifier key.

Optimizing these code editing triggers for touchscreens requires adaptations of their presentation and user interaction. For example, drag-and-drop operations might be easier to perform than keyboard shortcuts or selection from long linear lists. Hence, the user study has aimed at examining ways in which interaction through gestures and multi-touch can replace conventional WIMP methods.

Supporting refactoring commands through direct manipulation is challenging because individual steps of the process may require additional parameterization. For that purpose, desktop IDEs typically display modal dialog boxes and configuration *wizards*. Due to their modal nature, however, they obstruct the underlying user interface and interrupt the programmer's workflow. In touch-based environments, parameters could be encoded into gestures so as to reduce modal UI elements. In addition, parameters could be pre-defined with sensible default values, which would eliminate the need for any configuration in advance.



(a) Eclipse (b) Xcode

Figure 4.3: Refactoring *Application Menus* in Eclipse (left) and Apple's Xcode (right). Since refactoring commands are added over time, the menus keep growing until items must be nested into multiple sub-menus. Some of the commands were assigned arbitrary keyboard shortcuts by the IDE developers.

Since refactoring is a central part of code editing, the user study has included frequently used commands for *restructuring* code. The following sections introduce this development practice in more detail.

4.2 Refactoring

The process of *refactoring*, that is, increasing the quality of programs so that future changes are easier to perform and less time-consuming, is closely linked to software maintenance. Opdyke [Opd92] has defined refactoring as "behavior preserving program transformations". Fowler et al. [Fow99] have later described refactoring as the process of restructuring software without changing its externally "observable behavior". In other words, when programmers *refactor* code, they attempt to improve the internal structure of software but keep the exposed functionality identical for consumers of the

software. (The term “consumers” includes not only end users of the software but also other programs that might use its services.)

Due to the high percentage of maintenance-related work, refactoring code is a frequent and important activity. For example, Zhenchang and Stroulia [XS06] have reported that up to 70% of the structural changes of the Eclipse IDE source code can be attributed to refactoring. Furthermore, the *Extreme Programming (XP)* methodology, widely used among professionals, advocates refactoring as continuous activity during development.

Without the practice of refactoring, software quality tends to degenerate over time as source code is continually modified because of changing requirements, bug fixes, or addition of new features. In his seminal paper “Programs, life cycles, and laws of software evolution” [Leh80], Lehman has formulated these observations as follows (first two laws of his eight laws):

1. “An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.” (Lehman categorized constantly evolving software for solving real-world problems as “E-type” programs.)
2. “As a program is evolved its complexity increases unless work is done to maintain or reduce it.”

Fowler et al. [Fow99] have cataloged a number of refactorings in their book, including detailed step-wise instructions for each code transformation. The following two source code listings (Listing 4.1 and 4.2) show a simplified example for the transformation caused by the refactoring *Extract Local Variable*. (The example uses JavaScript since this language was used for the user study.)

```
1 function askDeepThought() {
2   return 6 * 7;
3 }
```

Listing 4.1: Function before refactoring.

```
1 function askDeepThought() {
2   var answer = 6 * 7;
3   return answer;
4 }
```

Listing 4.2: Function after applying the refactoring *Extract Local Variable*.

At the source code level, the transformation is obvious: The multiplication has been *extracted* into the local variable `answer`. With the introduction of a descriptive variable name, the intent of the code may have become clearer while the resulting behavior of the code has not been modified. (The clarification of intent is, of course, hard to argue for such a simple example without context.) Figure 4.4 illustrates how the transformation works at the AST level. This AST format is based on the *Mozilla Spidermonkey Parser API*[1], but the structure is essentially similar for other parsers.

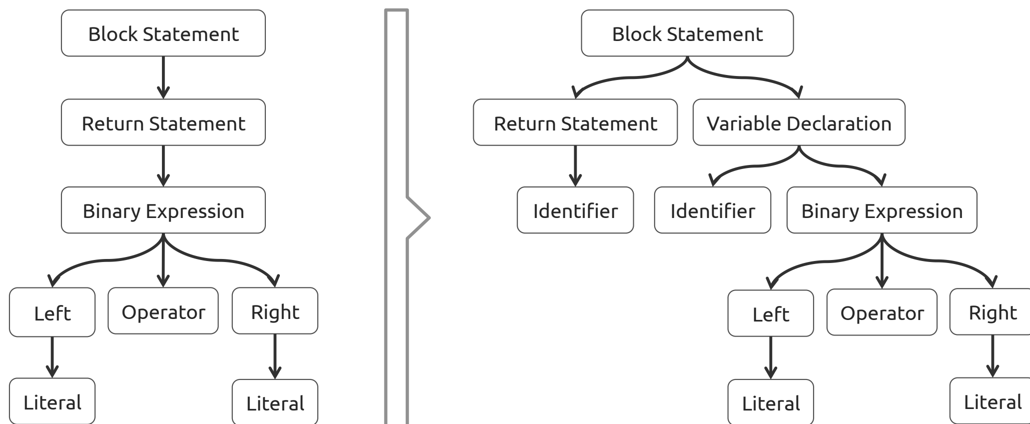


Figure 4.4: Transformation of the AST for the refactoring operation *Extract Local Variable*. (Inspired by Figure 13 in [NCV+12].)

A developer could have triggered this refactoring by selecting the multiplication and then invoking the command *Extract Local Variable* via menu selection or a keyboard shortcut. The IDE would then provide a UI element for entering the new variable name (e.g., a dialog box containing an input field). After confirmation of the change, the AST will be transformed according to the illustration, and the user could continue working with the updated textual representation of the transform. Formal analysis can partially

[1] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API`

guarantee that the behavior of the code is preserved, although formal methods are harder to realize for dynamic programming languages. Unit tests are another way of ensuring correct program behavior after refactoring.

Other structural changes require a more complex series of steps. For instance, users might first need to review the changes to be performed and make adjustments before confirming the operation. If the user ignores potential warnings before executing the refactoring, the result might lead to errors and thus to code that does *not* result in the same behavior as before.

In order to better understand the benefits and drawbacks of refactoring, and to make sound decisions for tool design, researchers have examined *how* developers refactor. Murphy-Hill et al. [MHPB09] have extensively studied developers' refactoring practices and noticed two frequently employed main strategies: *floss refactoring* and *root-canal refactoring*. The authors describe the two tactics as follows:

> "During floss refactoring, the programmer uses refactoring as a means to reach a specific end, such as adding a feature or fixing a bug. Thus, during floss refactoring the programmer intersperses refactoring with other kinds of program changes to keep code healthy. Root-canal refactoring, in contrast, is used for correcting deteriorated code and involves a protracted process consisting of exclusive refactoring."

In other words, developers steadily apply *floss refactoring* to improve the code they are currently editing; they may even be unaware of refactoring taking place. With *root-canal refactoring*, on the other hand, developers may deliberately set time aside for performing a larger structural change. While the former tactic is proactively interleaved with normal programming, the latter is explicitly carried out when deteriorated code has accumulated after a number of changes. Due to their differences in time and context of execution, both strategies need adequate support with tooling.

4.2.1 Refactoring Tools

Studies have shown that the refactoring tools of modern desktop IDEs are underused [MHB07, MHPB09]. Developers perform up to 90% of all transformations manually [MHPB09]. Since manual refactoring can be slower and more error-prone than relying on tools, research has investigated the reasons why developers avoid existing IDE facilities. Among other reasons, such as developers not being aware of the provided

IDE commands, researchers have primarily blamed the lack of usable interfaces and shortcomings concerning interaction [MHPB09].

Figure 4.5 shows the confirmation dialog for the refactoring *Change Method Signature* in the Eclipse IDE. The developer made problematic changes in the previous configuration dialog and now has to review the listed errors by going back to readjusting the refactoring parameters. Alternatively, he may choose to cancel the refactoring or to continue and thereby introduce potential issues into the program.

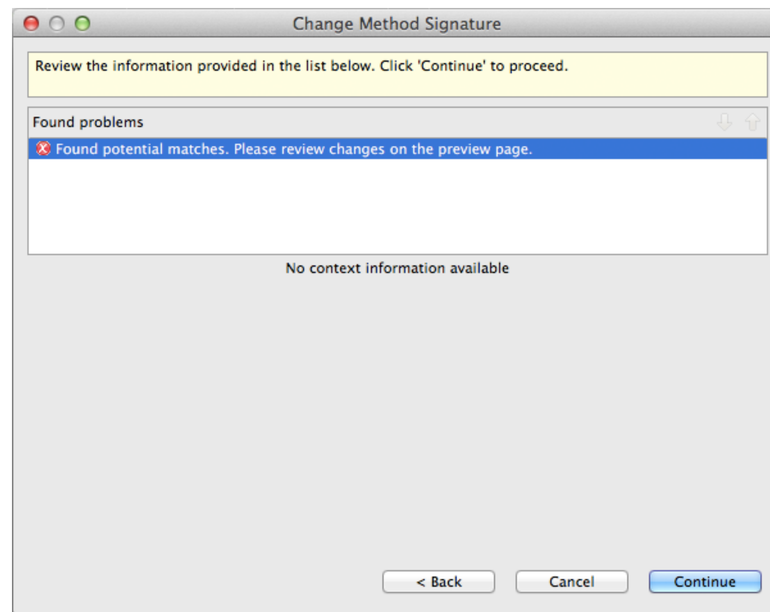


Figure 4.5: Confirmation dialog with listed errors after triggering the refactoring *Change Method Signature* in the Eclipse IDE.

Figure 4.6 shows another error message after invoking the command *Extract Constant*. The error message indicates that the developer's previous selection of code is incorrect. Although the error message vaguely points out the problem (an expression must be selected), the developer has to confirm the dialog, re-select code, and try again.

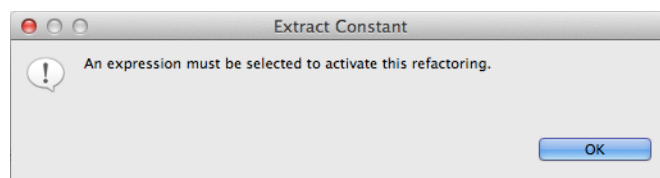


Figure 4.6: Error dialog after triggering the refactoring *Extract Constant* in the Eclipse IDE.

These errors point towards a number of design issues with the built-in refactoring commands of IDEs [MHB08]:

- It is often unclear which parts of the code must be selected in order to correctly invoke the command. Also, the mapping between command name and refactoring behavior is non-intuitive.
- Some refactorings first show modal configuration dialogs with parameters that need to be adjusted or reviewed.
- Vague and confusing error messages force developers to go back to configuration dialogs, readjust parameters, and try again.

The combination of unclear invocation, obtrusive configuration dialogs, and vague error messages might be part of the reason developers often resort to manual restructuring. Murphy-Hill and Black [MHB08] have identified and described these issues in more detail and designed techniques to assist developers in refactoring. For instance, they have developed *Selection Assist* (colorizing of whole statements) and *Box View* (simplified view of nested statements) to facilitate selecting the correct part of source code prior to invoking the command; *Refactoring Annotations* display feedback as colored overlays and render arrows when certain preconditions are violated.

Furthermore, Murphy-Hill and Black [MHAB11] have devised techniques for improving the initiation phase of a transformation. They have argued that having to recall the (sometimes imprecise) names of specific refactorings and their invocation through long menus or arbitrary keyboard shortcuts aggravate the triggering of commands. As alternative mechanism, they have used *marking menus* (see Chapter 3) and thus exploited spatial mappings by assigning commands to their most suitable item direction in the circular menu.

Refactoring by *drag-and-drop* [LCJ13] appears as natural and logical next step to improve the user interaction of refactoring. Since drag-and-drop is a lightweight method for combining both the initiation and configuration of a command into a single action, some of the overhead associated with performing an action can be reduced. Parameters can automatically be inferred by the source and target of the drag-and-drop action, or they are simply set to reasonable default values. For example, as demonstrated by Lee et al. [LCJ13], the exact location of newly created code for the *Extract Method* refactoring is automatically determined by the drag target. In contrast, such precise control is not supported by regular Eclipse refactorings.

Lee et al. have measured the physical effort of drag-and-drop refactoring in a user study. They have found that, compared with traditional methods relying on menus and keyboard shortcuts, the technique reduced the overall effort with respect to all required keystrokes, mouse clicks, and mouse movements.

4.2.2 Gesture-driven Tools

As stated earlier, gestures have the advantage of encoding multiple parameters into a single, fluid action. Rather than enforcing indirect interaction with code through menus and keyboard shortcuts, touch-enabled devices let users directly transform code using finger motions.

Previous research on developers' preferences for restructuring code may also be well-aligned with the use of gestures. Vakilian et al. [VCN+12], for instance, have concluded, "programmers prefer lightweight methods of invoking refactorings, usually perform small changes using the refactoring tool, proceed with an automated refactoring even when it may change the behavior of the program, and rarely preview the automated refactorings". Moreover, Murphy-Hill et al. [MHPB09] have observed that developers rarely change the configuration of refactorings, suggesting that default values are adequate in most cases; extra interface elements for adjusting parameters may thus be unnecessary.

Assuming that simple gesture designs are used, touch interaction implicitly results in "lightweight interaction" since the developer's fingers already touch the screen for other actions such as scrolling. Moving the fingers to the code structures of interest and performing a local transformation seems most natural. This interaction form also improves command triggering because developers are not burdened by navigating long menus and freed from remembering keyboard shortcuts. Since most code changes are local and small, the physical effort required should be within acceptable ranges.

Without reviewing configuration and preview dialogs, programmers might temporarily set the code into an invalid state if the modification violates particular conditions. However, since errors and warnings are usually ignored when restructuring software [VCN+12], behavior-preservation can be accomplished *after* the transform, for example by means of manually reviewing the change, applying suggested error fixes, or executing unit tests.

In their project *Hands-on Math* (see Chapter 3), Zeleznik et al. [ZBAK10] have demonstrated how gesture-driven transformation enhances interaction in a mathematical application. Among other things, they have shown how dragging with one finger and pinching with two fingers reorders and simplifies terms in mathematical equations. Although such modifications are equally conceivable for source code, this approach entails some challenges. It raises questions such as the following:

- How does the interaction scale to the large number of existing commands for editing code?
- How are commands without obvious mappings to gesture-driven interaction best supported?
- How are conflicts and ambiguities arising from other existing gestures (e.g., standard gestures of the operating system) resolved?
- How can additional parameterization and input be realized when the gesture itself cannot encode all information?

To summarize the previous sections: Refactoring is an important and frequent activity, but existing tools insufficiently support the process. Developers prefer lightweight, unobtrusive, and configuration-less tools for assistance when practicing *floss refactoring*, that is, frequent and small changes during programming to counteract code degeneration. Direct manipulation through gestures seems promising for code editing on touchscreens, but the details of the interaction need further study. By means of a conducted user study, I attempt to explore these details in the following sections.

4.3 User Study

The aim of the study was to examine how users apply multi-touch and gestural interaction to source code editing and restructuring on a touchscreen. Since interaction through a pen instead of finger motions has been shown to be beneficial in related application domains [ZBAK10], participants had to perform all tasks using both interaction styles. The study has revealed preferences for pen and finger input and given insights into their respective performance characteristics. The results can be used as guidelines for implementers of touch-enabled code editors.

This user study is different from previous studies on touch-centric code editing since it has investigated non-constrained editing in a regular text-based programming language. Prior work (Chapter 3) has either studied desktop IDEs (e.g., Eclipse), different domains (e.g., maths), or structured editing using visual programming techniques. In addition, users directly worked on a touchscreen in a specially prepared environment for code editing.

Involving users into the design of suitable gestures, an approach commonly called "participatory design", might lead to better results than gesture designs by developers and experts. Wobbrock et al. [WMW09] have remarked that gestures designed by developers do not necessarily reflect user behavior since developers are driven by technical issues revolving around reliable recognition. Moreover, Wobbrock et al. have found that their own gesture set covered only 60.9% of the gestures designed by users. These findings indicate that users generate valuable solutions that should be considered when incorporating gesture-driven interaction into new domains.

4.3.1 Editor Operations

The tasks of the study build on a set of standard editor operations. Users had to perform gestures for all operations listed in Table 4.1. This list is based on multiple sources:

- Commands of the *edit* and *refactor* menus of popular development environments such as Eclipse, Visual Studio, Xcode and Sublime Text.
- Recent research of refactoring practice, yielding commands that are regarded as important and commands that are frequently executed by developers [KZN12, NCV+12, VCN+12].
- Informal feedback from software practitioners (colleagues and friends).

Although the list of operations should not be regarded as exhaustive, both qualitative feedback from participants during the study and results from the post-study questionnaire seemed to suggest that no essential and frequently used commands had been missing in the study.

The operations are categorized into five different groups: Text editing operations, selections operations, code editing operations, refactoring operations, and navigation operations. While I will analyze selection operations in depth in Chapter 5, this study

Category	Operation
Text Editing Operations	Move Caret Copy/Paste Undo/Redo
Selection Operations	Select Identifier Select Multiple Identifiers Select Line Select Multiple Lines Select Block
Code Editing Operations	Move Lines Duplicate Line Delete Line Toggle Comment
Refactoring Operations	Extract Method (Without Locals) Extract Method (With Parameter) Inline Method Inline Temp Replace Temp With Query Introduce Explaining Variable (Extract Local) Rename (Multiple Variables)
Navigation Operations	Goto Method Declaration

Table 4.1: All 20 operations used in the study: 3 text editing operations, 5 selection operations, 4 code editing operations, 7 refactoring operations, and 1 navigation operation.

already contains basic commands for selecting identifiers, lines, and blocks. Many editing commands and refactorings require valid selections, users therefore had to perform the selection action either as part of the tasks or in isolation.

Code intelligence operations (see Figure 4.1) were *not* part of this study: first, because code intelligence functionality is often used during code *creation* (see Chapter 6); second, because incorporating additional operations would have further increased the number of tasks for the participants. Since all tasks had to be performed twice (once using the pen and once using the fingers), concerns about the total study time per participant limited the final number to twenty operations.

The reasoning behind including at least one navigation operation (*Goto Method Declaration*) was to see how participants would perform actions where code editing involves scrolling the editor viewport or jumping to different parts of the source code.

4.3.2 Participants

All participants filled in a questionnaire (see Appendix A) before the test. They were asked to specify their experience in certain programming languages, IDEs, and their usage of devices with touchscreens.

16 participants (14 male, 2 female) from the University of Regensburg, aged between 21 and 32 years ($M = 24$), were recruited. All participants were studying as undergraduate, postgraduate, or PhD students in computer science disciplines such as media informatics or information science.

While all but one of the participants indicated (on a 5-Point Likert scale) that they use devices with touchscreens *always* or *frequently*, 9 stated that they *never* use a pen as input device. All participants were right-handed.

12 participants had between 2 and 5 years of programming experience, 2 had more than 10 years. 11 participants regarded themselves as *quite experienced* in the programming language Java, 4 selected *very experienced*. As for JavaScript, 7 participants indicated *quite experienced* and 4 indicated *very experienced*. 10 participants considered themselves as *quite experienced* in using the Eclipse IDE, 2 were *very experienced*.

In addition, the participants named the programming languages and IDEs in which they regarded themselves at least *somewhat experienced* (number of mentions in parentheses): PHP (8), C++ (7), C (5), C# (5), Visual Studio (5), NetBeans (4), and Objective-C (3).

4.3.3 Test Setup

The test system consisted of two main components: an editor running on an *iPad 3* tablet showing the source code for the tasks, and a second connected editor running on the laptop of the experimenter. By means of a socket connection between the two systems, all touch events on the tablet and the keypress events of pen buttons were visualized as overlays on the experimenter's editor (Figure 4.7).

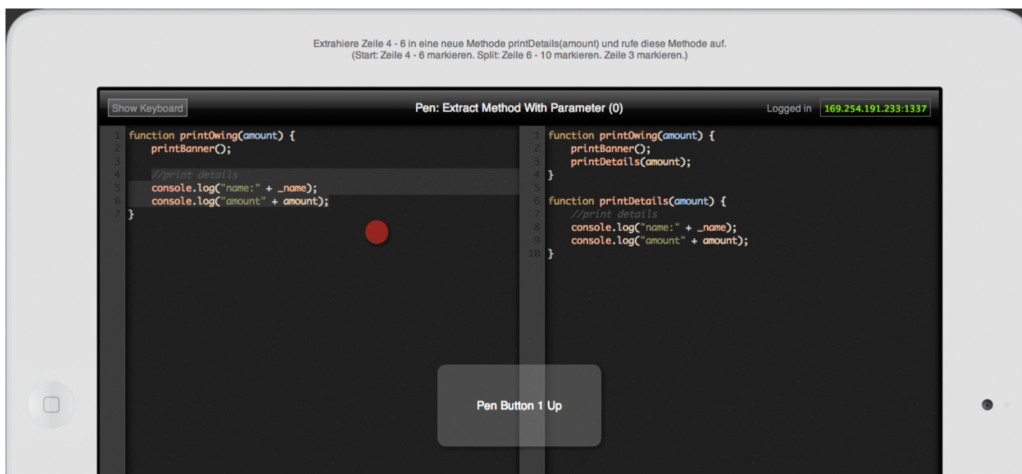


Figure 4.7: The experimenter's system showing the split editor window with a surrounding tablet frame. A touch point (red circle) and pen button press (semi-transparent gray box) are displayed as overlays in the editor area. The overlays visualize the participant interacting in real-time on the tablet and enable screen recording of the entire session on the laptop. Instructions on the top of the screen assist the experimenter in introducing each task.

The system of the experimenter not only passively visualized and recorded the interaction, but the experimenter could also actively act as "wizard" and control particular editor functionality of the tablet instance. The following interactions were directly reflected in real-time on the participant's tablet editor:

- Modifications of the source code.
- Selections of source code lines.
- The cursor position.
- Scrolling of the editor viewport.
- Showing or hiding the on-screen keyboard.

A *split view* on both systems showed the initial state of the source code on the left side and the desired state on the right side. In order to ensure that all participants received the same instructions, additional notes for each task were displayed on the experimenter's system. These instructions also included information about which lines to select or where to position the cursor (see Appendix A).

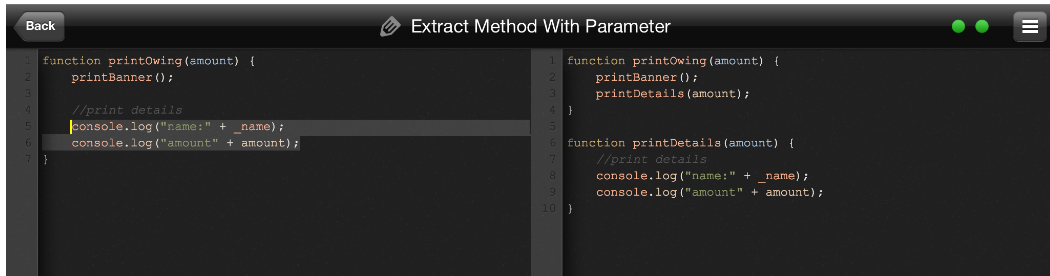


Figure 4.8: The participant's system showing a split editor instance on the tablet. The left side shows the initial state of a code example; the right side shows the desired state. The editor also reflects the actions of the experimenter's system on the laptop (e.g., selecting lines to introduce a task).

The entire setup (Figure 4.9) resembled, to some extent, a "Wizard-of-Oz" experiment, although all participants were fully aware of interacting with a remotely controlled system. (In conventional "Wizard-of-Oz" studies, participants are lead to believe that they are interacting with a real, working system.) Using this setup, the experimenter could track all tablet interaction on the laptop. Since each code example could be explained to the participant by highlighting particular code lines or positioning the cursor, inconvenient pointing on the small tablet screen in front of the participant could be avoided (Figure 4.8).

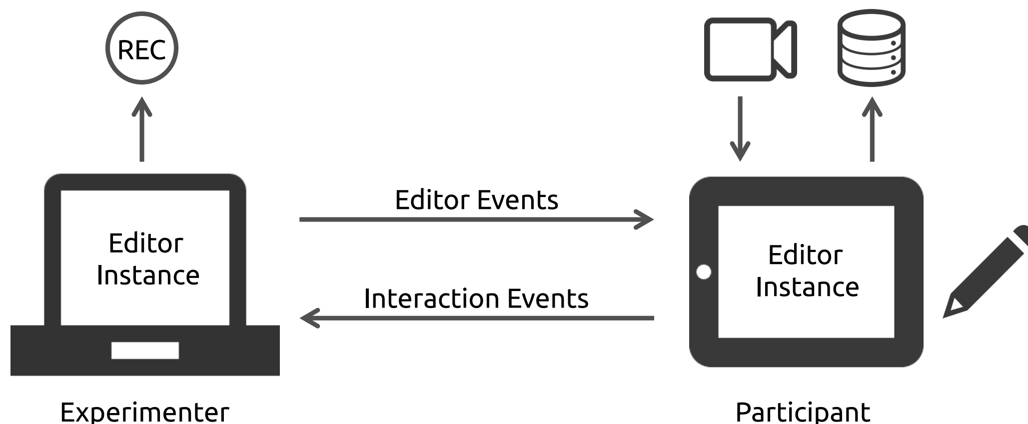


Figure 4.9: Illustration of the test setup: The experimenter's laptop sends editor events to the participant's tablet and receives interaction events from the tablet. All data is recorded and logged into a database.

The pen used in this study was an *Adonit Jot Touch* with two hardware buttons and a transparent touch-disk attached to the pen tip. The transparent disk ensures that the

tablet correctly registers the touch, and it minimizes the occlusion problem of the pen tip; hence, users could still see the source code characters under the pen tip through the disk.

For later analysis, all interaction events were logged to an SQLite database on the tablet. Since not all characteristics of the interaction can be reconstructed from logged touch events, the area around the tablet was captured on video so that the participant's hands and pen usage could be seen.

In order to reduce the participants' mental load for refactoring tasks, the choice of programming language fell on JavaScript. Due to JavaScript being a dynamically typed programming language, the participants had to concern themselves less with issues regarding the type system. Rather than being distracted by types of variable declarations or return types, the users should concentrate on the *interaction* of transforming the source code.

4.3.4 Procedure

The procedure itself was primarily based on the "guessibility study" by Wobbrock et al. [WMW09]. They have achieved good results by first showing users the *effect* of a surface gesture and then letting participants perform the *cause* of the gesture. Since the test system did not respond to user input and accepted all input, the users' behavior was not affected by technical aspects such as gesture recognition.

In this study, the participants were first introduced to the test setup and could then try a demo task. Each of the twenty different tasks had to be done once using the pen and once using normal touch interaction without the pen. Consequently, each participant completed forty tasks in fully randomized order; in total, 640 tasks were performed:

16 participants x 20 tasks x 2 input types = **640 tasks**

Participants took 75 minutes on average for the test (including filling in the pre-study and post-study questionnaires).

A single task consisted of the four phases:

Instruction Phase
: The experimenter introduced the code example using the previously mentioned features of the test setup. He made sure that the participant understood both the initial state of the source code and the desired state.

Preparation Phase
: The participant, thinking aloud, should try to find an adequate gesture. He was free to touch the surface or use the pen device.

Articulation Phase
: When the participant was ready to articulate the gesture again, he could start recording by pressing on the task title. Another press stopped recording when the participant was satisfied with the result.

Evaluation Phase
: A modal dialog displayed two post-task questions that the participant was asked to answer before moving on to the next task.

Similar to the study in [WMW09], the first question of the evaluation phase asked if the participant thought the performed gesture was a "good match for its intended purpose" (factor *goodness*, measured on a 7-Point Likert scale). As for the second question, the SMEQ (Subjective Mental Effort Question) recommended by Sauro and Dumas [SD09] was used. Users should indicate their perceived effort by moving a slider on a scale ranging from "not at all hard to do" to "tremendously hard to do" (values from 0 to 220, respectively). This scale has been shown to be reliable and easy for participants to use in its interactive version.

After all tasks had been performed, the participants filled out a final questionnaire where they indicated which input method they prefer (pen, fingers, or both) and which commands they frequently use in their development environments.

4.3.5 Results

The following sections report results regarding the amount of agreement among the participants, the relationships between performance measures and answers to the post-task questions, and qualitative data from questionnaires and observations.

Figure 4.10: After each task, participants answered two questions. They rated the perceived *goodness* of their performed gesture and set the perceived effort on a scale (SMEQ) using a slider.

Agreement

To classify all performed gestures, both the video captures and the logged touch events have been analyzed. Analyzing the videos included manually decomposing gestures into their key components and encoding that information for statistical tools. Hence, this process involved human judgment about which gestures consist of similar parts and should consequently be viewed as the same gesture for a task.

While the videos show how participants touch the surface and move their fingers or the pen, an additional custom-built visualization component of the test system displays the complete *gesture trail* (Figure 4.11). This component was exploited as assistive tool for validating the video-based judgments. For each task, it displays all finger touches, all pen touches, or all finger *and* pen touches. Each option applies to either a single participant or all participants.

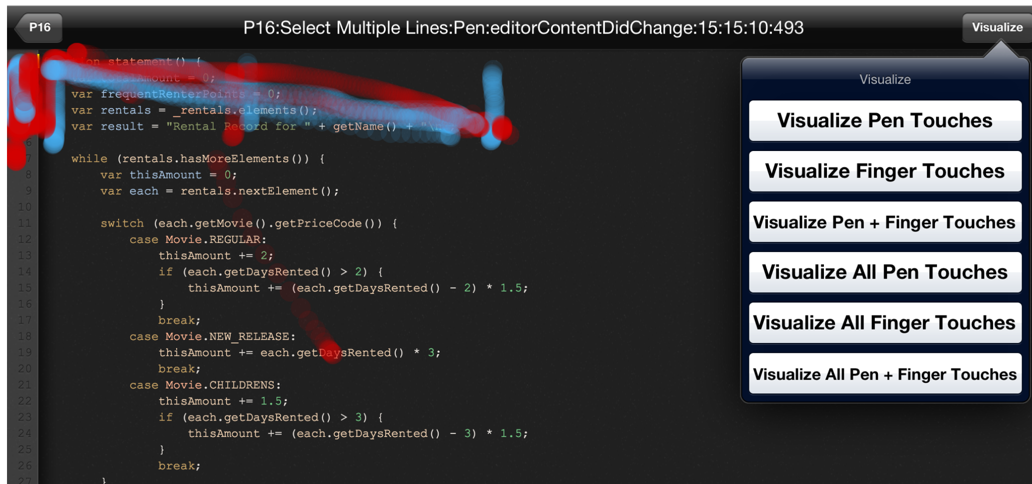


Figure 4.11: Visualization of the test system: Different options display finger touches and pen touches, either per participants or for all participants. Here, two visible patterns of gesture trails show the agreement for the task *Select Multiple Lines*.

The option to display all combined touches of a task creates a visual impression of the agreement among all participants: The more overlap of semi-transparent gesture trails, the higher the agreement. Scattered trails with less overlap signify less agreement. Figure 4.11 shows an example of a strongly pronounced effect with two visible patterns: One group of participants selected multiple lines of code by swiping over the line numbers in the gutter on the left side of the editor, while the other group swiped across the code block from top-left to bottom-right. Figure 4.12 illustrates the effect in weaker form for the *Undo/Redo* task: Although some trails approximate to circular motions, other patterns resemble swipe motions. Overall, the agreement was therefore lower for this task.

To calculate the agreement scores, the approach adopted from [WMW09] has been used. For example, analyzing the task *Select Line (Pen)* resulted in five groups of size 8, 5, 1, 1, 1. The agreement score is calculated as follows (Equation 4.1):

$$A_{slp} = \left(\frac{8}{16}\right)^2 + \left(\frac{5}{16}\right)^2 + \left(\frac{1}{16}\right)^2 + \left(\frac{1}{16}\right)^2 + \left(\frac{1}{16}\right)^2 = 0.36 \tag{4.1}$$

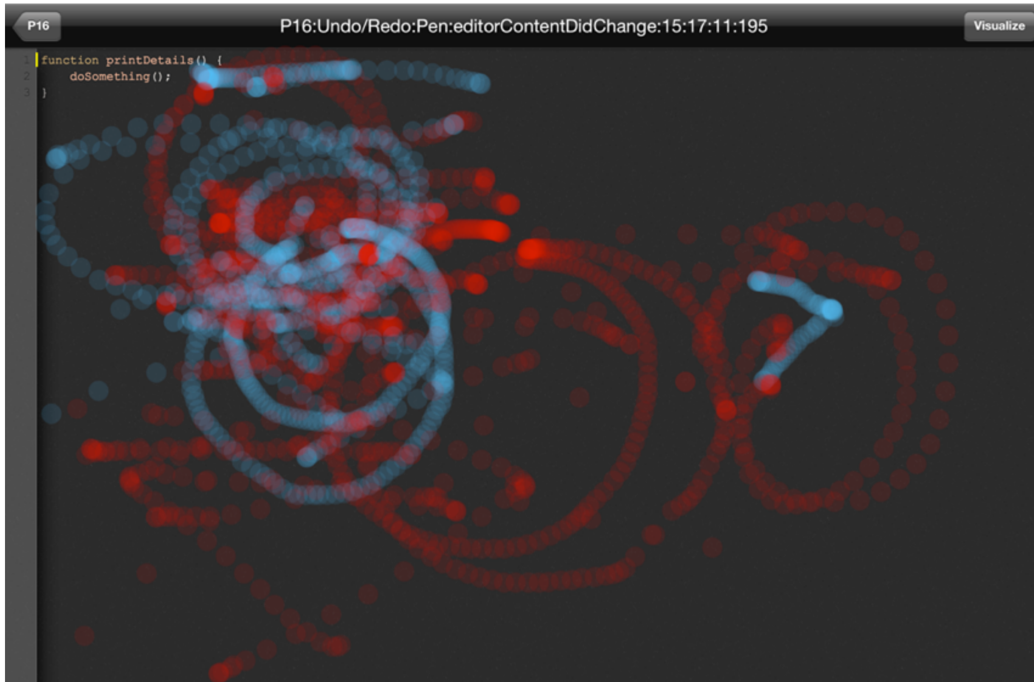


Figure 4.12: Visualization of the agreement for the task *Undo/Redo*: Although overlapping circular patterns are recognizable, the overall effect is less pronounced.

For the task *Move Lines (Finger)* with group sizes of 12, 1, 1, 1, 1, the agreement score is calculated as follows (Equation 4.2):

$$A_{mlf} = \left(\frac{12}{16}\right)^2 + \left(\frac{1}{16}\right)^2 + \left(\frac{1}{16}\right)^2 + \left(\frac{1}{16}\right)^2 + \left(\frac{1}{16}\right)^2 = 0.58 \qquad (4.2)$$

According to these scores, the gesture for moving lines led to higher agreement ($A_{mlf} = 0.58$) than the gesture for selecting multiple lines ($A_{slp} = 0.36$). Figure 4.13 graphs the agreement scores for all tasks and both input methods. As expected, the agreement scores are lower for refactoring tasks involving multiple steps than for basic operations such as selecting one or more lines of code. Overall, the agreement scores are lower ($M = 0.20, M_{Pen} = 0.19, M_{Finger} = 0.20$) than in [WMW09], which might be due to the more complex application domain of this study. Users generally agreed most on selection gestures for identifiers, lines, and blocks, and on gestures for moving the caret and moving lines.

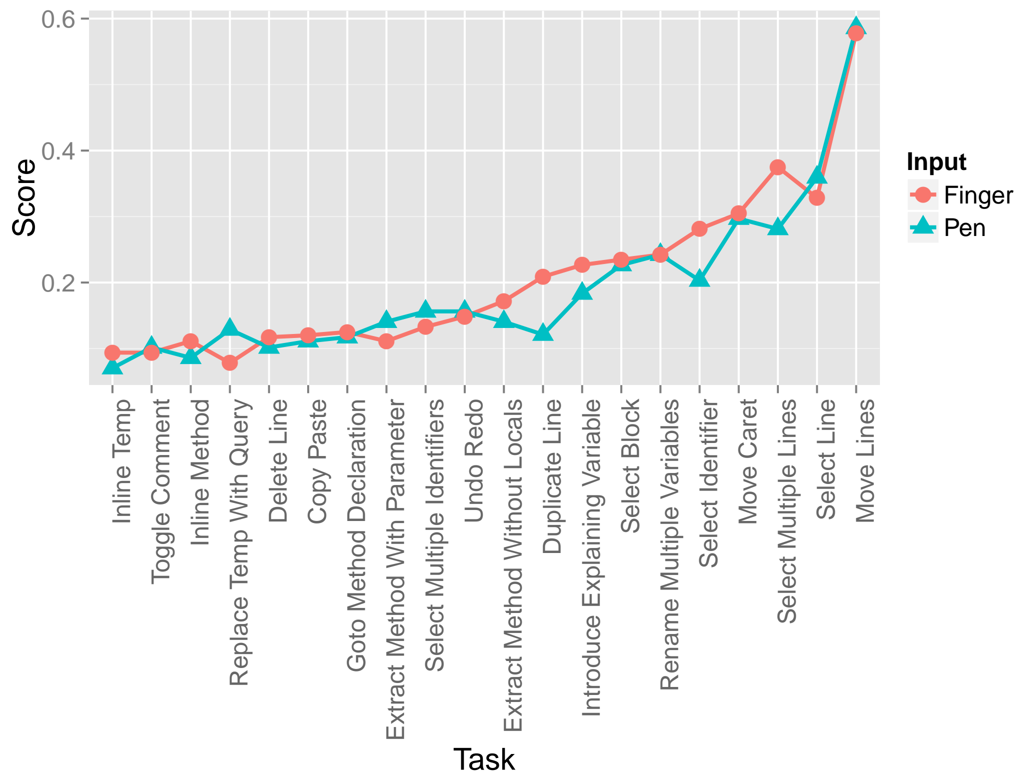


Figure 4.13: The gesture agreement scores for all tasks.

Goodness, SMEQ, and Task Times

Goodness ratings have a mean value of 5.21 ($M_{Pen} = 5.21, M_{Finger} = 5.22$); SMEQ values have a mean value of 50.25 ($M_{Pen} = 49.34, M_{Finger} = 51.16$). The values for goodness and SMEQ negatively correlate: Higher goodness rankings tend to have lower SMEQ values and vice versa ($r = -0.83, t(38) = -9.26, p < 0.01$). The mean task times (in seconds) for the preparation phase (Prep) and articulation phase (Art) are: $M_{Finger,Prep} = 65.9, M_{Finger,Art} = 21.7, M_{Pen,Prep} = 65.6, M_{Pen,Art} = 24.3$

Relationships: Goodness–SMEQ–Agreement–Articulation-Time

The relationships between the two post-task values for goodness and SMEQ, the calculated agreement score, and the measured articulation time are illustrated in the two bubble charts of Figure 4.14 and Figure 4.15. The diagrams show that the most agreed upon gestures were those that users perceived as good matches and least

effortful. Also, those gestures were articulated fastest. This is contrary to some of the results in [WMW09] where articulation time did not affect goodness ratings, and gestures that took longer to perform were perceived as easier.

Furthermore, the results for the number of touch events are not in line with the findings in [WMW09]: In this study, gestures consisting of more touch events were perceived as more effortful (but did not have lower goodness ratings). However, previous results could also be confirmed: Better gestures became more quickly apparent to participants (less preparation time) and popularity (high agreement) identified better gestures.

No significant differences between pen and finger interaction were detected in any of the mentioned values.

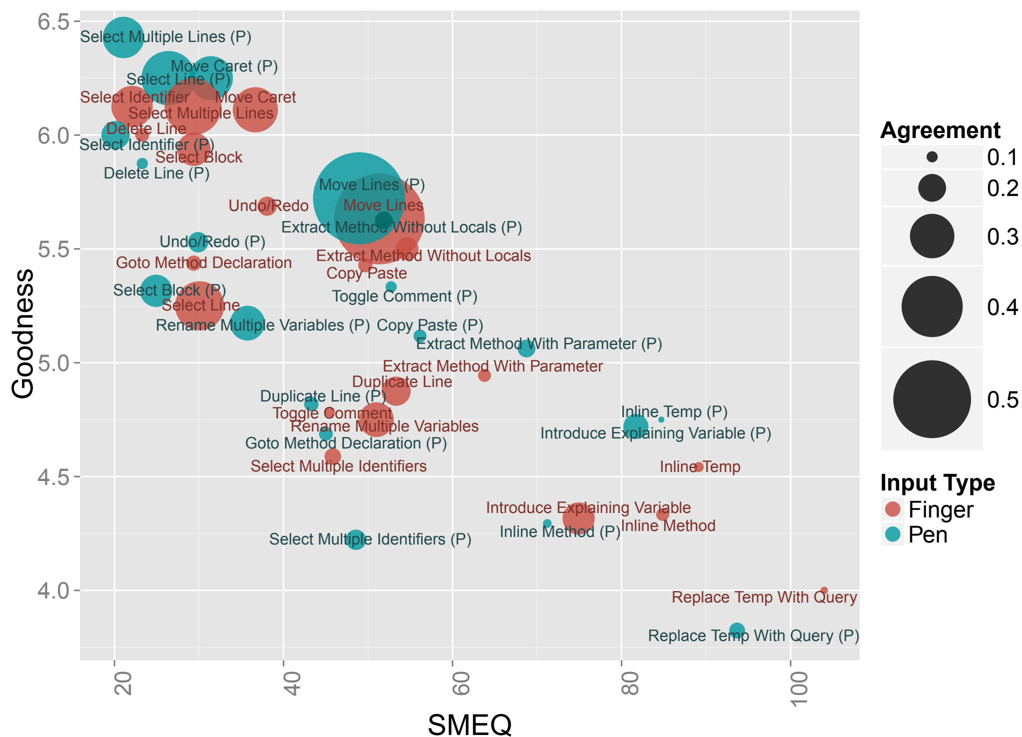


Figure 4.14: Bubble chart showing aggregated values for gesture goodness (vertical), SMEQ (horizontal) and agreement (size).

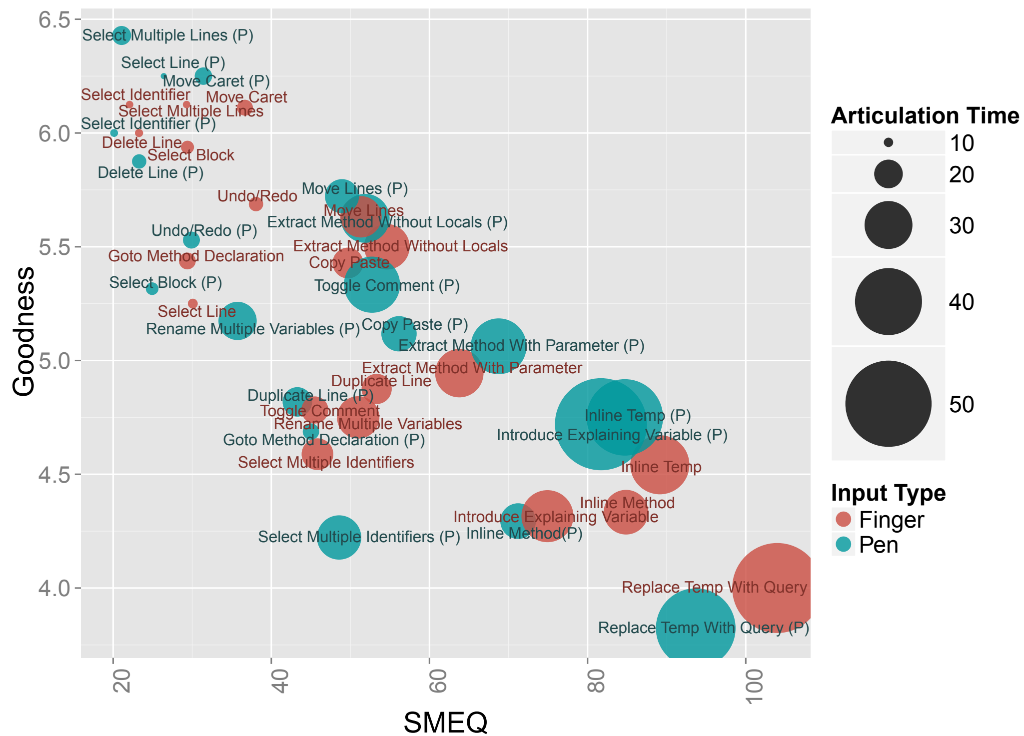


Figure 4.15: Bubble chart showing aggregated values for goodness (vertical), SMEQ (horizontal) and articulation time (size).

Input Preferences and IDE Features

The results from the post-study questionnaire (see Appendix A) indicate the participants' preferred input methods and the IDE features they frequently use. 44% of the participants chose the pen as their preferred input method, 25% chose finger touches, and 31% preferred mixed pen and finger interaction.

According to their own judgment, participants frequently employ the following IDE features (number of mentions in parentheses): Rename (6), Auto-complete (5), Navigation to method or class (5), Auto-format (4), Save (3), Extract method (2), Create new method (2).

4.3.6 Observations

The users' mental models appeared strongly influenced by the interaction concepts of mobile operating systems. Most of the participants could easily be identified as

"Android users" or "iOS users": They either directly referred to a specific platform (e.g., "On Android, I can...") or mentioned certain platform-specific features (e.g., the magnifying glass on iOS).

Participants frequently asked for context menus since they either could not think of a suitable gesture or found a menu more convenient in particular situations. At the same time, however, they expressed their dislike for menus that contain too many items; consequently, they took the expected frequency of operations into account when deciding if an operation should be added to a context menu.

Some participants were concerned that selection and gesture recognition might not be precise enough in a working system. They supposed that inaccuracies might lead to frequent re-selections and adjustments in the editor.

Most users seemed to prefer one-handed gestures and only conservatively employed multi-touch interaction. Hence, only few gestures were performed with more than two fingers. According to comments during the study, the pen was perceived as more accurate than interaction using finger touches. Participants frequently decided to perform the same gesture for both the pen and finger version of the task. The two hardware buttons of the pen were sometimes used as replacements for the left and right buttons of a mouse.

As far as specific refactoring operations are concerned, users generally seemed to find it easier to *extract* than to *inline* code. Some of the *inline* operations resulted in sequences of unnecessary steps for completing the task. For users without prior knowledge of inline refactorings, it was not apparent that the transformation could be automated and hence only required a gestural trigger.

4.3.7 Discussion

As previously stated, the mean agreement score was lower than in the study by Wobbrock et al. [WMW09]. The difference could by explained by the dissimilar user group (programmers) and application domain (software development). This assumption is further supported by the finding that faster gestures had higher goodness ratings and lower SMEQ scores. Participants probably valued the efficiency of execution more than in the Wobbrock study where users "were recruited from the general public and were not computer scientists or user interface designers" [WMW09].

Some operations led to only little agreement on an adequate gesture (as also reported by Wobbrock et al.). On the one hand, the time-consuming design of this type of study might be disproportionate to the expected outcome. On the other hand, a small number of operations yielded high agreement scores. The majority of participants, for example, proposed using the line gutter to select code blocks. Without involving users, this area might not have been predicted as the most popular target for line selection.

During the study, some participants stated that they would retrospectively solve individual tasks differently or that they could not remember their previous actions. Since the tasks were presented in randomized order, participants might have had problems developing more consistent gesture sets. These issues could be reduced by providing users with the full set of tasks in advance and then letting them freely design the corresponding gestures.

Another limitation of user-elicited gesture designs is their tendency to lack creative and novel multi-touch operations. In this study, most users confined themselves to one-handed interaction using only one finger. In another study on custom gesture sets, Oh and Findlater [OF13] concluded, "Our findings showed that even when asked to create novel gestures, participants tended to focus on the familiar."; and Zelenik et al. [ZBAK10] remarked, "In essence, if the effort expended on bi-manual interaction appears to greatly exceeds any performance benefit gained, then unimanual interaction may be preferred." However, the lack of innovation also generates positive effects: Conservative gesture designs might be easier to learn for users, and more effortless to implement for developers.

In the pre-study questionnaire, 56% of all participants responded that they never use a pen for touch input. In the post-study questionnaire, 44% chose the pen as their preferred input method, and 31% preferred mixed pen and finger interaction. Overall, this distribution indicates that the pen was positively received. (Users viewed the device as more accurate.) Although the pen with its hardware buttons provides additional flexibility for interaction design, its integration into a system for code input poses challenges: Pen usage could be inconvenient when code editing operations frequently involve keyboard usage.

Other challenges arising from user-elicited gestures include the difficulties of ensuring non-ambiguous and conflict-free gestures. Since the users worked on isolated tasks in a non-responsive system, they could not foresee all potential problems of their proposed

solutions. Some of the solutions might interfere with operating system gestures or learned standard gestures that should not be repurposed. The resulting issues and possible solutions for resolving conflicts in a responsive system are further discussed in Chapter 7.

The work presented in this chapter can be extended into several directions:

Scaling to more operations
The proposed gestures contain solutions for only a basic subset of possible operations. However, the more operations an application has to support, the harder it is to find non-conflicting custom gestures; hence, in addition to gestures, applications may need to display touch-optimized menu (see Chapter 7).

Supporting *inter-file* operations
The code examples of the tasks included only *intra-file* operations. It remains open to explore how commands have to be adjusted so that multiple source files are part of a single interaction.

Enabling disambiguation
Some operation might benefit from temporary disambiguation options. This would allow the same gesture to be reused for multiple operations (e.g., through interaction techniques such as *Under-the-Rock Menus*, see Chapter 3).

Improving discoverability
Since custom gestures are not self-revealing (i.e., users do not know where and how to perform gestures in the UI), additional disclosure mechanisms should inform users about the available operations and the details of their invocation.

4.3.8 Design Recommendations

Figure 4.16 illustrates the final gesture set, which could serve as starting point for implementers of touch-enabled code editors. Gestures that are likely to interfere with conventional platform gestures have been disambiguated by adding a second touch point. Refactoring operations, for example, might otherwise cause conflicts with built-in gestures for scrolling the viewport. Motions and targets, however, have not been modified to maintain the identities of the proposed gestures.

Also, the set includes user interface elements that should be considered as interactive zones: To select multiple lines and code blocks, for instance, the majority of users have

Figure 4.16: Gesture set including basic operations for selection, editing, and refactoring in text-based editors. (SelectFirstIdentifier > 2FingerTap * means: Select the first identifier, then (>) perform multiple (*) taps using 2 fingers.)

employed the line gutter as target for swipe motions. As another subtle, yet important, usability aspect, the study has revealed the need for additional "buffer zones" at the top and bottom of the editor area: Almost all participants accidentally touched buttons in the top navigation bar when they tried to perform their gestures in the editor area below.

4.3.9 Conclusion

Rather than radically changing software development tools for touchscreens, I suggest enhancing existing text-based editors with gestural interaction for code editing and refactoring operations. In the first part of this chapter, I have introduced the different types of operations and their impact on the representation of code. In the second part, I have discussed *refactoring* in more detail since this software development practice is a crucial and frequently performed activity during code editing.

The third part has presented a user study on how participants applied gesture-driven interaction to standard code editing tasks. In contrast to arbitrarily designed gesture sets, user-generated gestures are based on the most agreed upon solutions among all participants. The agreement among participants can be measured and serves as basis for the final set of recommended code editing gestures. This participatory approach was supported by a custom-built test setup consisting of two connected code editor instances. That way, participants could concentrate on their tasks, whereas the experimenter remotely controlled the participant's tablet editor to explain the tasks and record all interaction events.

Furthermore, the study explored the advantages of using a pen (instead of finger touches) for interaction. Although the comparison has not revealed any significant differences in the measured performance values, qualitative feedback has hinted at the value of the pen's accuracy. However, for the rest of this work, I will not further consider pen interaction due to the issues associated with code *entry* (Chapter 6).

Beyond the results, I have learned two lessons from conducting the study: First, the approach can be time-consuming, both with regard to the required test setup and the analysis. Hence, the question whether the expected output justifies the cost should be considered before conducting similar studies. Second, when the proposed gestures are integrated into a working system, conflicts might arise due to interference with the existing multi-touch commands of the operating system. Neither the experimenter

nor the participants can easily foresee these (often subtle) implementation issues. Nevertheless, the study has uncovered a number of valuable opportunities for source code interaction. Gesture-driven UI widgets could be integrated alongside pure gestural approaches to reduce the potential for ambiguities and conflicts. Concrete solutions to that end are demonstrated in Chapter 7.

Chapter 5

Selecting Source Code

Before programmers perform a code editing operation, they usually *select* textual structures of the source code. Since selection commands are so frequently performed, touch-centric editors should provide efficient interaction techniques while compensating for the absence of a physical keyboard and mouse.

In this chapter, I introduce the mechanics of code selection by examining desktop editors and text editing applications of mobile platforms. Furthermore, I report the findings of a user study that has revealed frequently performed operations and selection patterns in a realistic software development situation. The study results have motivated the design of a touch-enabled *syntax-aware* selection technique and a number of gesture-driven and widget-based methods for selecting *structural code regions*. These techniques are presented at the end of this chapter.

5.1 Introduction

For most code editing operations, selection is a precondition to set the target or scope of an operation that the programmer is about to execute next. For example, a simple *copy* command requires the programmer to specify which part of the source code he wants to copy, whereas the corresponding *paste* command is either performed by inserting the clipboard content at the cursor position or by replacing an active selection. The following sections introduce the selection concept of desktop applications and current

mobile platforms. Also, I highlight the differences between selecting in “Emacs-type” [Fin91] editors and modal editors such as VIM.

5.1.1 Terminology and Selection Mechanics

A selection marks a defined *range* of the text as “active”, that is, highlighted for the user to distinguish selected code from non-selected code. In the code editor, a selection sets an internal state to be considered for subsequent editing operations. Selected text consists of a start position and an end position. When users select text beginning at the start position, they can extend the *range* to a position before or after this start position. It is perhaps more accurate to call the positions *selection anchor* and *selection head*, respectively: The anchor is first set and remains at a fixed position; the head extends the selection to either before or after the anchor. (Other text editors use similar terminology: Emacs, for example, calls the mentioned components *region*, *mark*, and *point*). Figure 5.1 illustrates the anchor and head of an active selection range (the body of the method `askDeepThought`).

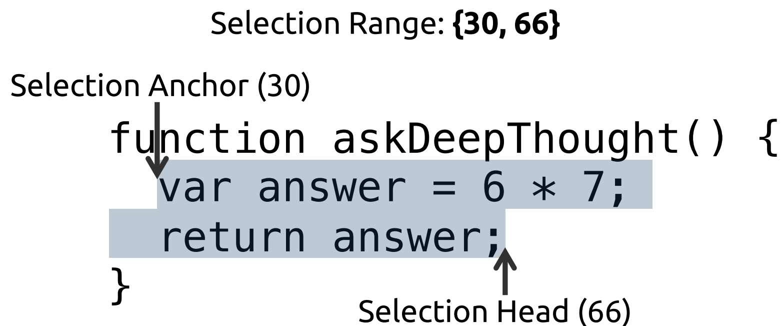


Figure 5.1: Illustration of the selection *range* from the *anchor* at character index 30 to the *head* at index 66.

When a selection is active, most operating systems provide that entering a character replaces the entire content of the selected text with the new character. When no selection is visible, the range can be thought of being *empty*. For instance, a cursor positioned at character index 66 would result in a range from character index 66 to index 66; the anchor and head are at the same positions. Also, some editors support a feature called *multiple selections*, that is, different parts of the source code are in selected state and can then be modified at once. With multiple active selections,

one range becomes the *primary selection*, which is the sole receiver of operations not compatible with multiple active ranges.

5.1.2 Modeless vs. Modal Selection

The basics of the selection concept were introduced with Tesler's editor *Gypsy* about 40 years ago (see [Tes12] and Chapter 3). The selection features, as today implemented by IDEs and general-purpose text editors, have arguably been influenced by Emacs. The command set of "Emacs-type" editors [Fin91] enables *modeless* selecting of textual structures. By combining selection operations with cursor movement commands (e.g., jumping to the end of a word or line), users can select larger structural regions with little effort. This interaction style is in contrast to the *modal* editing of editors such as VIM, where users either explicitly enter a special selection mode or perform their selection implicitly in "normal mode" (i.e., without visually marking text).

Selecting text constructs in VIM, for instance, first involves entering a *mode*, then (optionally) entering modifiers, followed by commands for *text objects* (e.g., words, sentences, or paragraphs) or *motions* (e.g., end of word or forward one line). A sequence of keys forms a composed keyboard shortcut that can precisely perform the intended operation; however, the functionality of individual keys and their flexible interplay have to be learned and practiced. The acquired skills cannot be easily transferred to other editors that use different or simpler mechanics. Moreover, the modal interaction style may be difficult to transfer to touchscreens. The advantage of touch-based selection, as later demonstrated, can be attributed to the ability of users directly pointing at textual structures. This direct manipulation style might be thwarted by modal editing. The following discussion, therefore, primarily revolves around the more familiar Emacs-type selections.

5.1.3 Selection in Desktop IDEs

Although IDE manufacturers differ in concrete implementations of selection features, the central functionality is largely identical across editors.

Most keyboard shortcuts consist of one or two *modifier* keys (*Command [Mac OS]*, *Control*, *Option [Alt]*, *Shift*) and a character key. Similarly, mouse selection can be altered by pressing additional modifier keys. While the OS provides general keyboard

Action	Positions	Element
Move/Select to	Previous Next	Character
Move/Select to	Start of current Start of previous End of current End of next	Word
Move/Select to	Start of current Previous (Nearest Character) End of current Next (Nearest Character)	Line
Move/Select to	Start of current Start of previous End of current End of next	Paragraph
Move/Select to	Start of End of	Document
Select	Start to end of (all of)	Document
Move/Select to	Previous Next Enclosing	Syntactical Element

Table 5.1: List of common OS- and IDE-provided commands for cursor movement and text selection.

shortcuts that work across all text-based applications, IDEs may override these default shortcuts to enable more adequate code-centric behavior. For instance, the IDE might override the OS shortcut for selecting the next word with selecting the next *syntactical element*.

Table 5.1 lists a number of common text selection commands. Most commands are provided by the OS. IDEs add functionality or alter shortcuts to operate on basic syntactical structures. Most shortcuts can either be used for navigation (moving the cursor to a new location) or selection (moving the cursor to a new location *and* selecting the text to that location, usually by holding the modifier key *Shift*). On Mac OS, for example, holding the *Option* and *Shift* keys while pressing the *Right Arrow* key moves the selection head to the end of the current word (first press) and with subsequent presses to the end of the next word, and so on.

Text elements include characters, words, lines, paragraphs, and the current document. Some IDEs provide the feature to select an enclosing syntactical element (i.e., the next containing element). Multiple presses of the corresponding shortcut could select the following structures in a program written in *CamelCase* notation[1]:

```
[Cursor Position]
  > Next Camel Hump
    > Whole Word
      > Containing Expression
        > Containing Block
          > Whole Method Body
            > Whole Method
              > Whole Class
```

Table 5.2 lists standard OS selection commands when using the mouse. (In Windows, some of the commands are application-specific.) Pressing the mouse button, dragging towards a target location, and releasing the mouse button selects a *range*. Multiple *non-contiguous* ranges can be selected by holding *Command* (Mac OS) or *Ctrl* (Windows). Holding *Option* (Mac OS) or *Alt* (Windows) allows for selecting *rectangular blocks*. Double-clicking selects words, triple-clicking extends the selection to paragraphs.

Action	**Positions**	**Element**
Select	Contiguous	Range
Select	Non-contiguous	Ranges
	Rectangular	
Select	Start to end of (whole)	Word
Select	Start to end of (whole)	Paragraph

Table 5.2: List of elements that can be selected using the mouse.

5.1.4 Selection on Mobile Platforms

Since selection via the keyboard and mouse is not available on touch-enabled devices, smartphones and tablets provide on-screen widgets and gestures for text selection. In this section, I describe how the mobile platforms iOS and Android have attempted to solve the problem of precisely selecting text on small screens.

[1]http://en.wikipedia.org/wiki/CamelCase

The following description uses the example of the standard *Notes* app on iOS (as of version 7.1.2): Provided that the text was already set into the editable state by tapping into the text, selecting text first requires an *initiation gesture*. This gesture could be pressing-and-holding the finger onto the surface until a magnifying glass is displayed for accurate cursor movement (Figure 5.2, top-left). Releasing the finger invokes a pop-up menu containing selection buttons (Figure 5.2, top-right). Pressing the button "Select" selects the word under the cursor and replaces the menu with editing operations (Figure 5.2, bottom-left). Moving the selection handle on the left or right expands or shrinks the range under the magnifying glass (Figure 5.2, bottom-right). When the textfield is *not* in the editable state, pressing-and-holding directly transitions to the editing state, with the word at the touch location selected.

Figure 5.2: Initiating and changing text selection on iOS.

Text selection (including editing) on iOS follows a sequence of four steps:

1. Initiating the selection.
2. Changing the selection.
3. Performing edits.
4. Ending the selection.

In addition, users can exploit selection gestures (Table 5.3) to accelerate the transition to the selected state. For example, double-tapping selects the word at the location of the tap; two-finger-tapping selects the whole paragraph. Also, the selection handles snap to word boundaries when they are quickly dragged (swiped) towards the beginning or end of a word.

On Android (as of version 4.1.2), text selection is initiated by double-tapping or pressing-and-holding (called *long-press* on Android), which selects the word at the

Gesture	Element
Press-and-hold	Word
Double-tap	Word
Two-finger-tap	Paragraph
Swipe handle (to word boundary)	Range
Two-finger-drag (pinch/pull)	Range

Table 5.3: List of selection gestures and selected text elements on the iOS platform.

touch location (Figure 5.3, no. 2). In the selected state, a *contextual action bar* (Figure 5.3, no. 1) shows text editing options as buttons. Similar to the iOS selection widget, the Android handles at the start and end can be moved to adjust the range.

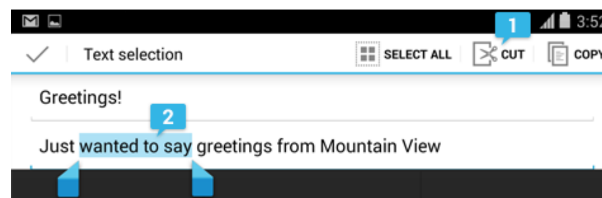


Figure 5.3: Initiating and adjusting text selection on Android[2].

Android, by default, does not support any text editing gestures (Table 5.4) except double-taps and long-presses. When the alternative *Swype* keyboard is enabled, the whole text can be selected by swiping from a special symbol in the lower left corner of the keyboard to the letter “A”. Furthermore, Android uses visually larger handles and does not employ magnification features for adjustments.

Gesture	Selection
Press-and-hold	Word
Double-tap	Word
Swipe to “A” (*Swype*)	Document

Table 5.4: List of selection gestures and selected text elements on the Android platform.

To improve on the approaches provided by mobile operating systems, researchers have proposed gesture-driven enhancements to text selection. For instance, Fuccella et al. [FIM13] (see Chapter 3) have developed a technique where gestures are performed on top of the keyboard: Two-finger-swiping to the left or right moves the selection head

[2] http://developer.android.com/design/building-blocks/text-fields.html

one word left or right; two-finger-swiping up or down moves the selection head to the start or end of the text.

5.2 User Study

In this section, I detail a user study that examined how programmers perform selections in an IDE. Results from the same study have been been published in [FHRC14], but focused on different aspects; that is, the paper has broadly described students' app coding behavior from different perspectives. In contrast, the findings presented here concentrate on the process of code selection and its analysis based on IDE interaction logs. The quantitative results of this analysis have contributed to devising appropriate interaction methods for touch-centric code selection. The concrete designs of these techniques are introduced in the last part of this chapter.

5.2.1 Participants

The data was collected during the final exams of two programming courses on Android application development at the University of Regensburg in the winter term 2012/2013 and the summer term 2013. The 78 participating students were majoring in media informatics or information science, typically studying in the second or third semester. (Both disciplines are closely related to applied computer science). Students had basic knowledge of object-oriented programming with Java from an introductory course taught in the first semester. Mobile application development with Java and the Android framework is a follow-up course taught in the second semester. Hence, all participants had prior experience with Java, the Eclipse IDE, and the Android Development Tools (ADT)[3]. The Eclipse IDE had been used as standard IDE for all Java courses; ADT provides Android-specific tooling as plug-in or standalone version.

5.2.2 Test Setup

The assessment was conducted in a university facility equipped with 57 personal computers running Windows and Linux operating systems. At each assessment, all

[3] `http://developer.android.com/tools/sdk/eclipse-adt.html`

students were divided into two consecutive groups in order to alleviate both technical and organizational supervision concerns.

Students were provided with a pre-configured version of ADT. Since they had used ADT for course assignments before, students were familiar with the features of the development environment. In addition to ADT, the following tools for logging and analysis of their coding behavior were pre-installed and automatically started at the beginning of the evaluation:

1. A screen recording software captured videos (for analysis of the students' problem-solving strategies).
2. A custom-built Firefox plug-in logged particular browser events (for analysis of the students' search patterns regarding documentation and help).
3. An Eclipse plug-in logged low-level IDE events (for analysis of the student's IDE usage and interaction).

The analysis of selections is based only on the log files of the third tool, the publicly available Eclipse plug-in *FLUORITE* [YM11]. Compared to other logging tools, this plug-in captures low-level commands in the editor so that researchers can explore frequently triggered commands and detect fine-grained usage patterns. FLUORITE logs all data into XML files, which are stored in the workspace when the IDE is closed. The authors of the tool have built a separate application for producing summary reports and charts. However, since their program did not provide the necessary information, all relevant data was extracted by parsing the XML files using custom scripts. Furthermore, a custom replay-application (presented below) supported more precise analysis of the performed selections.

5.2.3 Procedure

A handout (see Appendix B), given to students prior to the assessment, informed all participants that anonymized data would be collected for research purposes. Although they could choose to opt out of data collection, all students agreed to take part in the study.

All of the tools mentioned above were integrated into one single package, which was distributed over the local network before the assessment began. Students were then

asked to execute a setup-script that created desktop shortcuts for the IDE and browser. Additionally, the script started the video recording software.

After having configured their IDE workspace, students began working on the tasks that were described on extra handouts. Due to the limited exam duration of two hours (one and a half hour in the summer term), pre-installed workspace-packages provided students with existing code for extension or modification according to the requirements of each task. The following list briefly describes each task. (The tasks had primarily been designed by the lecturer of both Android courses, see Appendix B.)

Tasks in Winter Term 2012/2013

Quiz App
: Create the layout and UI code for a quiz app: The app contains textfields for the question and answer, as well as buttons for revealing the answer and going to the next question. The provided data model had to be used as foundation.

Responsive UI
: Implement a background operation: An existing app had to be extended so that an "expensive" operation (faculty calculation of a large number) runs in a background thread, displays the progress, and shows the result in a UI dialog.

Refactoring
: Refactor an existing app: The source code of the provided app had to be improved so that it adheres to software engineering principles and best practices that had been taught in the course (e.g., naming conventions or variable usage).

Tasks in Summer Term 2013

Tax Calculator
: Create the layout and UI code for a tax calculator app: The app contains a textfield for a price without tax and a textfield for the price including tax. When one of the textfields change, the other textfield should update correspondingly.

Debugging
: Debug an existing quiz app: The provided app contained three bugs that had to be uncovered. The causes of the bugs and the suggested fixes should be noted in a text file.

At the end of the assessment, a submission-script was started and compressed the workspace, log files, and video file into a single file. This file was then manually collected from each student. (The reasons for manual collection were due to file size restrictions imposed by the technical infrastructure.)

5.2.4 Analysis

As previously mentioned, the FLOURITE analyzer application did not prove useful for examining selection events. The XML files were thus analyzed using a combination of the following three methods:

- Scripts: Custom scripts were mainly used for generating tables of average and absolute command frequencies.
- Replay Application: A web UI read the XML files and replayed the entire session in an editor. This method was used for visual inspection of all events and sending information about individual events to a server application that returned extended data (see next section).
- Server Application: Selection events were sent to a Java application using the Eclipse framework for identification of selected AST nodes. This tool stored the results in a database for further analysis in statistics software.

This next two sections describe the functionality of the replay and server application in more detail, and explain how selections have been matched to AST nodes.

Selection Replay

The Eclipse IDE itself was *not* used for replaying the interaction logs due to the development overhead that this approach would have generated. The web-based approach allowed for quicker prototyping of the required functionality and omission of non-essential interaction events.

In order to reconstruct selections from the XML log files, all interaction events had to be replayed so that the current state of the source code could be inspected at a particular point of a participant's code editing session. For example, a selection event in FLOURITE is logged as follows (Listing 5.1):

```
1 <Command __id="91" _type="SelectTextCommand" caretOffset="253"
    end="301" start="253" timestamp="1073828" />
```

Listing 5.1: XML log format of a FLOURITE selection event.

To determine the selected part of the code based on the logged character indices, *all* previous events up to this point had to be replayed. Figure 5.4 shows a screenshot of the application that was built for that purpose.

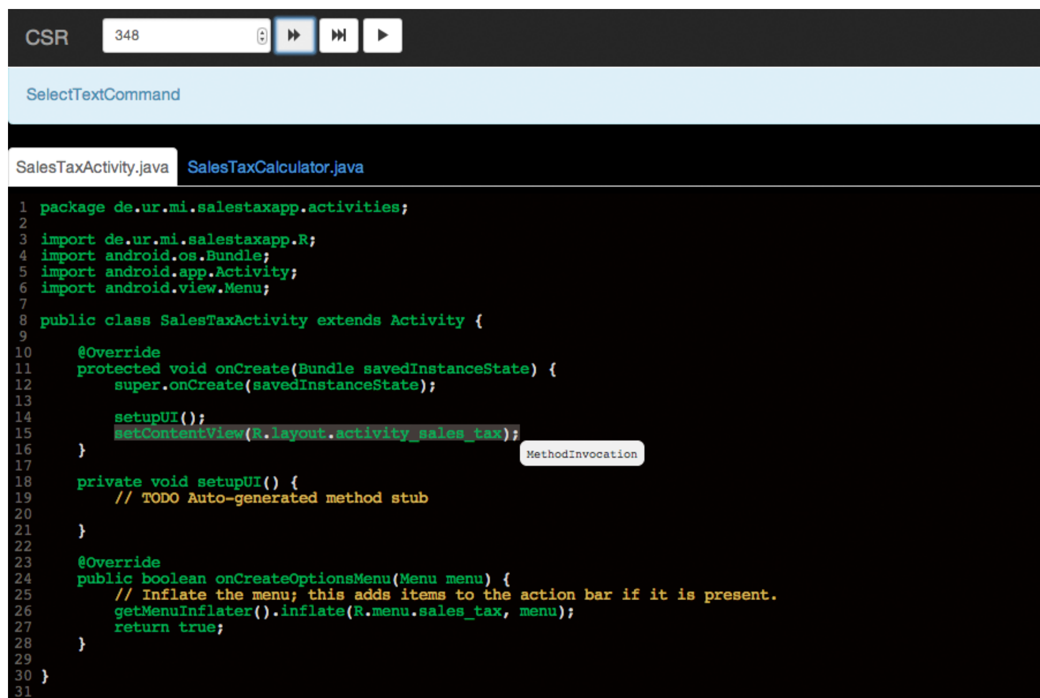


Figure 5.4: Web-based application for replaying and stepping through all logged events. Selection events are further analyzed by a server application.

After an XML file is dragged into the editor area, the control panel at the top of the screen enables stepping through single events of the session, fast-forwarding to an entered event identifier, or automatically replaying all events in real-time. The application replays only those events that modify the state of the source code in the editor, including events where participants switch to a different file-tab for performing edits. Also, the application displays all selections and sends information about the interaction to a server application for further analysis. Figure 5.5 illustrates the interplay between the client and server application. The system supports two different modes of operation:

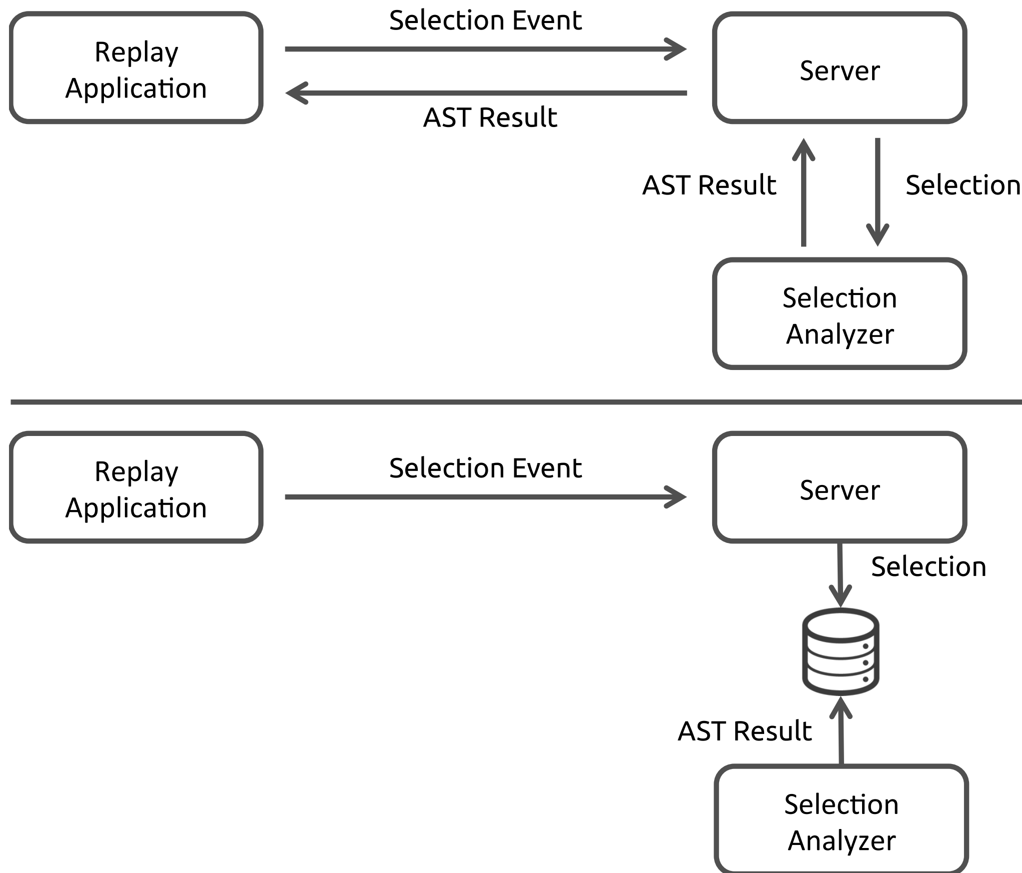


Figure 5.5: Components of the analysis system for selection events. Two modes are supported: Manual replay (top half) and automatic replay (bottom half).

Manual Replay

When manual stepping through events or the replay feature is used, the client sends information about the selection (the source, start index, and length) to the server. The server delegates the data to a Java application that determines the corresponding AST node for the selection and returns the result to the server. The server then sends the result to the client, which displays the information in a tooltip.

Automatic Replay

When all interaction events up to an entered event identifier are automatically replayed, timing-information is ignored so that the entire session is completed as quickly as possible. This mode first stores all selections in a database on the

server. After the storage phase, the Java application is launched, determines the AST data for all selections, and writes the results into the same database. The complete database is then processed by statistics software.

While manual mode was used for visual inspection of the selection process and single AST results from the server, automatic mode was exploited to bulk-process and analyze all selection events. Only Java source files were analyzed; other Android project files such as XML layouts or strings files were ignored.

AST Node Finding

The Eclipse JDT core framework exposes the necessary interfaces to find the AST nodes for a selection. The framework provides the `ASTParser` and `NodeFinder` classes: `ASTParser` creates a `CompilationUnit` instance based on a source string and configurable options. For the analysis of this work, the parser was configured to recover statements containing syntax errors. The resulting `CompilationUnit` instance (the AST root), the start of the selection, and the length of the selection are then passed into `NodeFinder`.
The node finding algorithm works as follows (as described in the documentation of `NodeFinder`[4]):

> "[F]irst the visitor tries to find a node with the exact start and length. [I]f no such node exists then the node that encloses the range defined by start and length is returned. [I]f the length is zero then also nodes are considered where the node's start or end position matches start. [O]therwise null is returned."

In other words, the algorithm first tries to find the first *covered node* (i.e., the node that is enclosed by the selection range) in a top-down traversal, and then falls back to finding the last *covering node* (i.e., the node that encloses the selection range).

5.2.5 Results

In this section, I report the results that have been generated using the previously mentioned tools for analysis. The results include:

[4] `http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/NodeFinder.html`

- The average and total frequencies of performed IDE commands.
- The types of selected AST nodes with their relative frequencies.
- The relative frequencies for matches at structural boundaries.
- The average number of selected lines.
- The relative frequency of selection directions.

Command Frequencies

Table 5.5 lists the average and total command frequencies.

Type	Average	Total
All Commands	1483.6	115717
All Unique Commands	23.0	112
Insert String	443.1	34558
Move Caret	313.4	24446
Delete (Backspace)	135.2	10549
Select Text	**108.3**	**8450**
Open File	91.1	7109
Movement (Arrow Keys)	72.3	5637
Content Assist Proposals (Automatic)	51.7	4029
Content Assist (Manual)	65.7	5127
Save File	31.8	2483
Paste	26.1	2035
Copy	15.3	1196
Cut	6.2	487
Quick Assist (Manual)	5.5	429
Undo	5.1	394
Save All (Menu)	5.7	446
Delete (Menu)	4.9	386
Organize Imports	4.2	326
Format Code	3.4	269
Rename	2.3	176

Table 5.5: List of the average frequencies (per participant) and total frequencies (across all participants) of performed Eclipse commands. The list contains only commands with avg. frequencies > 1.

On average, participants performed 1483 commands during the exam session. About 7% of all triggered commands are selection commands. Each participant executed, on average, only 23 unique commands. Excluding high-frequency operations such as inserting characters and moving the cursor, the average number of unique commands is even lower (14). This data shows that code selection is the most frequently performed operation after the (expected) dominating operations of character insertion, deletion, and cursor movement.

Relative AST Node Frequencies

Figure 5.6 lists selected AST node types after normalizing for the relative frequency.

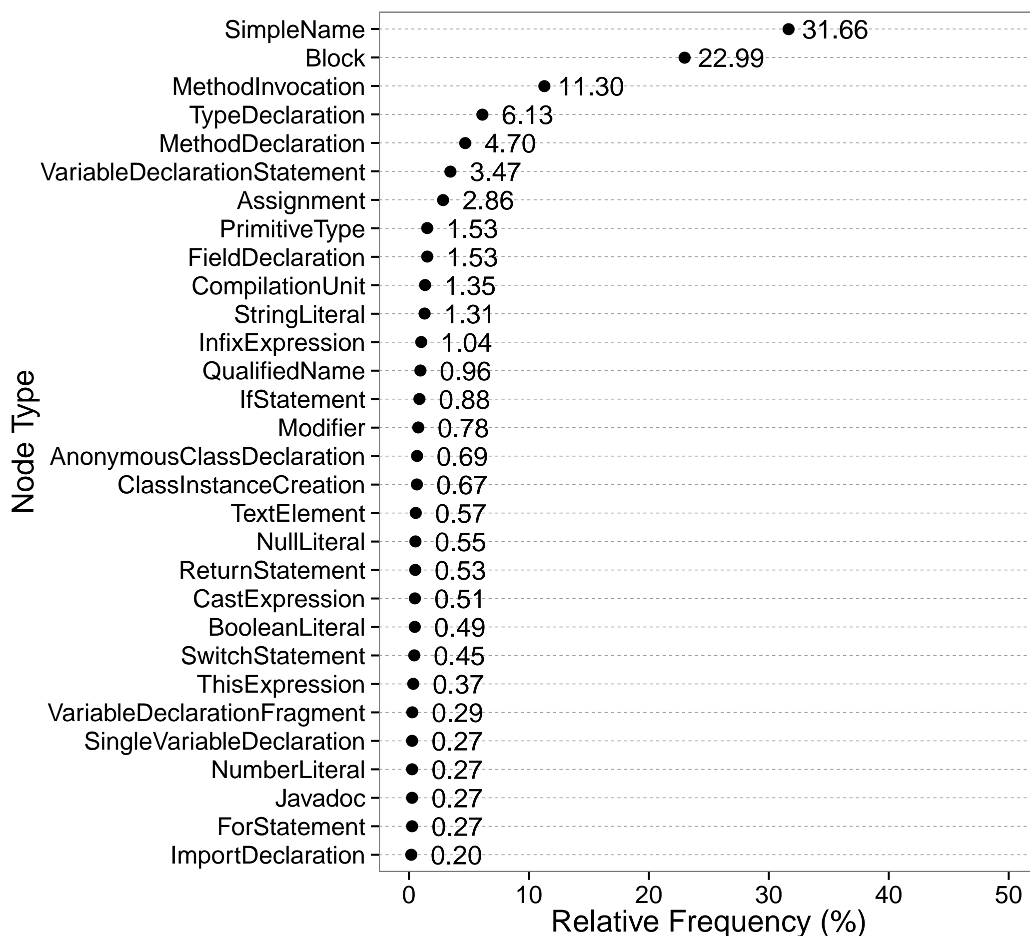


Figure 5.6: List of relative frequencies for the selected AST node types. The graph contains only frequencies > 0.2%.

The most frequently selected AST nodes (with relative frequencies > 10%) are names, blocks, and method invocations.

Structural Boundaries

Matching AST nodes for selections have been determined according to the described node finding process. To examine if the node boundaries precisely match the actual selection boundaries, the indices of the AST node (i.e., the start and end position in the source code representation) have been compared to the selection indices. As displayed in the chart of Figure 5.7, 51.1% of all selections match the AST boundary at the end position; 46.6% of selections match at the start position; for both positions, 40.4% exact matches have been detected. These results show that about half of all selections match at structural boundaries.

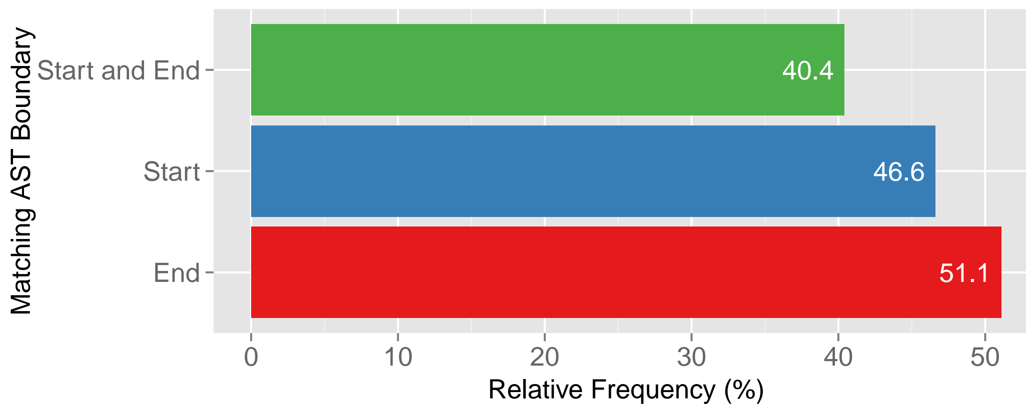


Figure 5.7: Relative frequency of selections precisely matching at AST node boundaries.

It should be noted that comparing node boundaries to selection ranges through an automated node finding process does not capture all matches. For instance, given the source code `priceWithoutTax.getContext()`, the user might have selected `getContext()`. Since the selection includes the parentheses of the method call, the node finding algorithm identifies a node of type *MethodInvocation*, with its start and end positions enclosing the entire source code; the node boundary and the selection boundary are not considered as matching. If the user had selected `getContext` without the parentheses, the node finding algorithm would have identified a node of type *SimpleName* and a match would have been recorded. Therefore, the results here should be regarded as conservative estimates.

Number of Selected Lines

Figure 5.8 graphs the number of selected lines for different AST node types. The average number of selected lines is highest for method invocations, type declarations, and method declarations. The unexpected result of method invocation nodes having a higher number of selected lines than method declarations can be explained by the use of certain Android design patterns: Developers frequently create anonymous classes for setting a click-listener on a UI object; this leads to the anonymous class spanning multiple lines as part of the method argument. Selections of variable declarations, field declarations, and comments span the lowest number of lines.

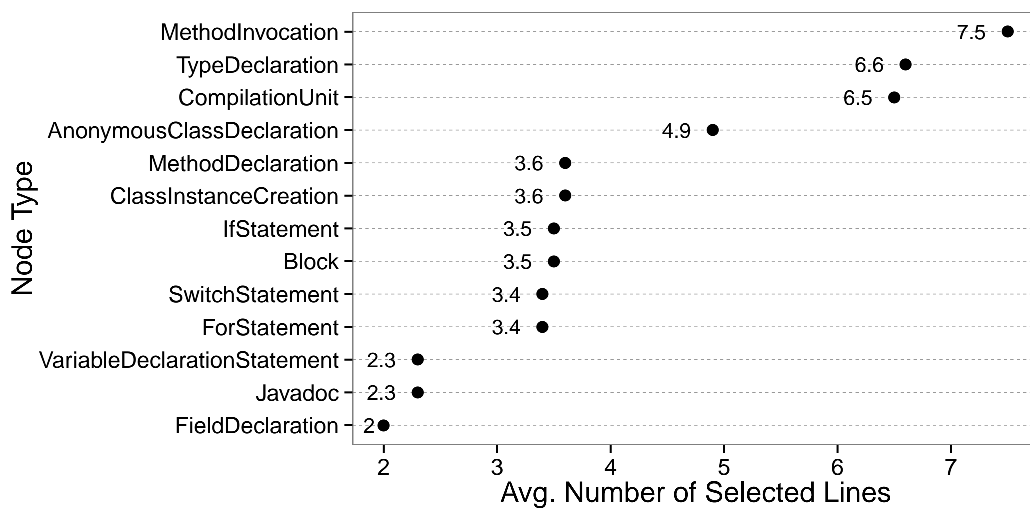


Figure 5.8: Average number of selected lines for AST node types. The list contains only selections spanning more than one line.

Selection Direction

52% of all selections were performed so that the selection head was positioned after the selection anchor (i.e., from left to right), whereas 48% were performed in the opposite direction. A goodness-of-fit test ($P(\chi^2 > 7.30) = 0.007$) shows that frequencies significantly differ from an expected equal distribution of 50% for each direction (which appears reasonable to assume).

Direction	Percent
Head after anchor (left to right)	52%
Head before anchor (right to left)	48%

Table 5.6: Percentage of selections for each selection direction.

5.2.6 Discussion

The user study has allowed for examining a large number of interaction events in a realistic development setting. Although most participants could be regarded as novices in programming and code editing, it stands to reason that results might be similar with more experienced programmers. In a prior study about structure editors, for instance, Ko et al. [KAM05b] have found that "name edits" were the most frequently performed editing operations (43% of their data) in a group of participants with "above-average" Java expertise. The study of this work supports their finding since AST *name* nodes were most frequently selected (relative frequency of 32%).

Furthermore, researchers previously observed the high number of backspacing in code editing environments (as compared with lower reported backspacing frequencies in regular document editing environments) [YM11]. This study confirms the finding. Reasons could be repeated fixing of typos, renaming, or rewriting code for new attempts at a solution. The low number of unique commands indicates that users tend to ignore a large number of available IDE commands.

The overall high number of selection events suggests that code selection is a worthwhile area for improvements, particularly because existing mobile platforms only have limited capabilities in this regard. The list with relative frequencies of selected AST nodes provides clues to possible optimizations. For instance, the selection of block statements or other frequently selected syntactical elements appears as promising target for enhanced interaction methods. Unfortunately, the employed logging plug-in FLOURITE does not capture *how* a user performs a selection. Differences between mouse and keyboard usage thus remain unclear. It seems reasonable to assume that more experienced programmers select code more often using keyboard shortcuts in order to avoid frequent switching to the mouse.

Irrespective of whether the keyboard or mouse was used, the selection directions appear broadly balanced, with slightly more operations initiated by placing the head *after* the anchor. As explained in the introduction, smartphones and tablets typically

display widgets (handles) for fine-tuning the selected range in both directions. As shown by the data, techniques that do *not* employ such widgets should keep supporting the initiation and extension of the selection range to both directions. Since the observed ranges have not exceeded eight lines, interaction techniques for one-handed two-finger selection of multiple lines should be feasible in most situations.

The results of this study have motivated the design of touch-centric selection mechanisms (presented in the next section) that consider structural boundaries, as opposed to treating source code as regular textual content. A large number of selections exactly match at structural boundaries. Hence, syntax-aware code selection, combined with gesture-driven interaction, could considerably improve the process of selecting regions of code on touchscreens.

5.3 Interaction Methods

In the following sections, I introduce the mechanics of *syntax-aware* code selection and present the details of the designed interaction techniques for code selection. All techniques have been implemented in the IDE shown in Chapter 7.

5.3.1 Syntax-aware Selection

Syntax-aware selection ensures that selection ranges align at the structural boundaries of code. This approach...

- reduces manually adjusting small selection handles.
- requires less touch-precision and therefore works with smaller font sizes.
- selects syntactic regions for subsequent editing operations (e.g., refactoring).
- combines both coarse and fine selection adjustments.
- supports multiple non-contiguous selection ranges.
- does not open or require the on-screen keyboard for selection.

Initiating the Selection

Selections are initiated by performing a touch-and-hold (long-press) gesture on the source code. This interaction is similar to the default behavior of text selection on mobile platforms. After a duration threshold, the source location at the touch point is determined. At this source location, the surrounding AST node (i.e., the innermost node that contains the source location) is searched. The selected range is then updated to match the boundaries of the found AST node.

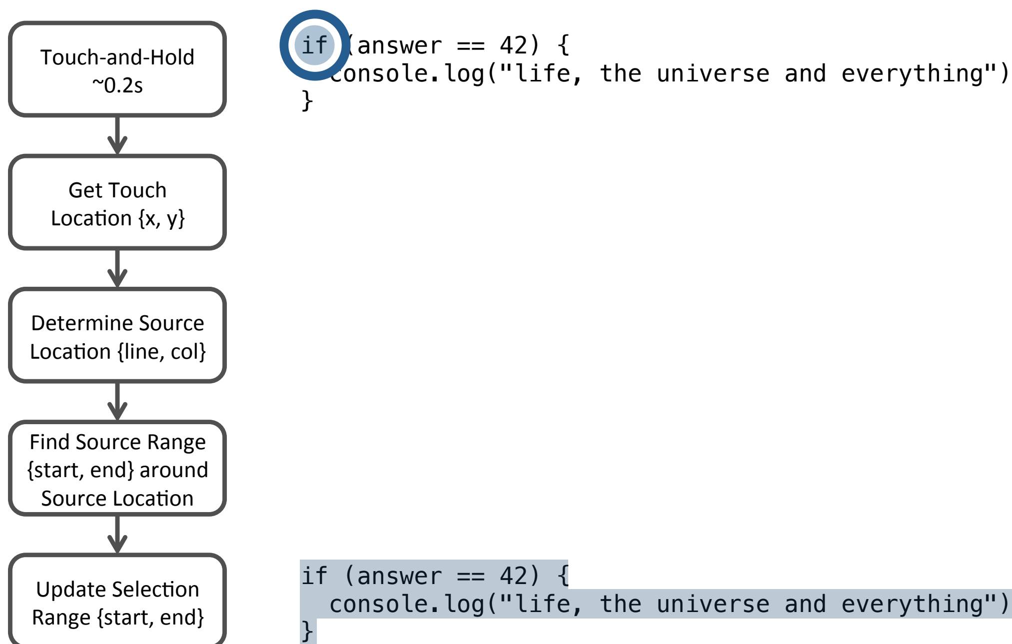


Figure 5.9: Main steps of initiating a syntax-aware selection operation in the code editor. The blue circle represents a duration threshold or *dwelling time* at the initial touch location.

In the example shown in Figure 5.9, the innermost surrounding node is of type *if statement*. (Unlike *concrete* syntax trees, *abstract* syntax trees do not record keywords or punctuation; touching-and-holding the keyword *if*, therefore, leads to the respective *statement* node being found.) Since this node consists of child nodes for the *test expression* and the *consequent statement*, these nodes are enclosed by the selection range. If the *if statement* contained an *alternate statement*, it would be included in the selection range. Since the same mechanics are applied for other AST node types, the regions that become selected should be quickly predictable for the user.

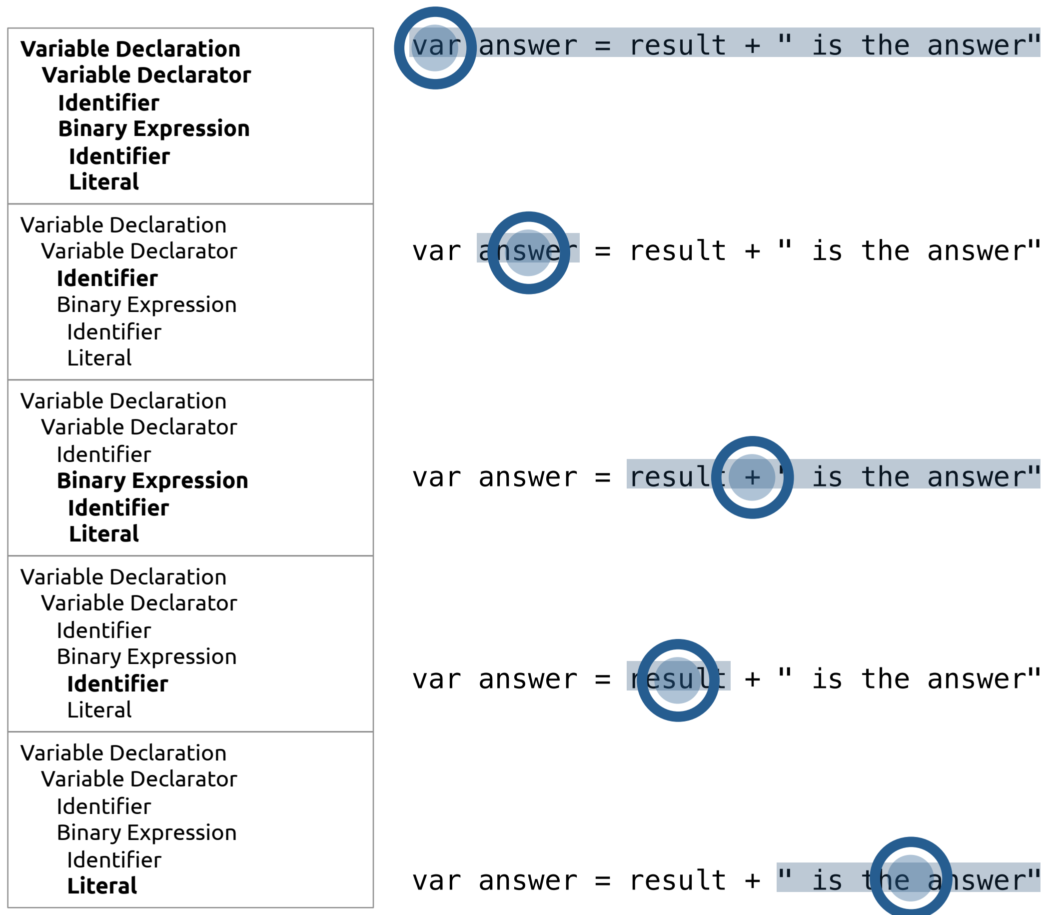


Figure 5.10: Touching different elements of the *variable declaration* node selects the corresponding child nodes in the syntax tree.

Another example is demonstrated in Figure 5.10. Here, different touch locations select different child nodes of a *variable declaration*:

- Touching the root of the declaration, the *keyword*, selects the entire declaration (for similar reasons why an *if* keyword selects the entire statement, see above).
- Touching the *identifier* of the *declarator* node selects only the *identifier*.
- Touching the *operator* selects the entire *binary expression*.
- Touching the left *identifier* node or the right *literal* node selects only these leaf nodes.

Technically, this approach depends on the concrete implementation of node *visitors* that traverse the syntax tree and either visit or skip certain child nodes. An alternative implementation, for instance, could skip the *identifier* node of a *variable declarator*. In the example of Figure 5.10, this would result in the additional selection of the *binary expression* node when the user performed a touch-and-hold gesture on `answer`. Skipping the touched child node selects the parent node, which may–depending on the node type–extend the range to before or after the touched node (Figure 5.11).

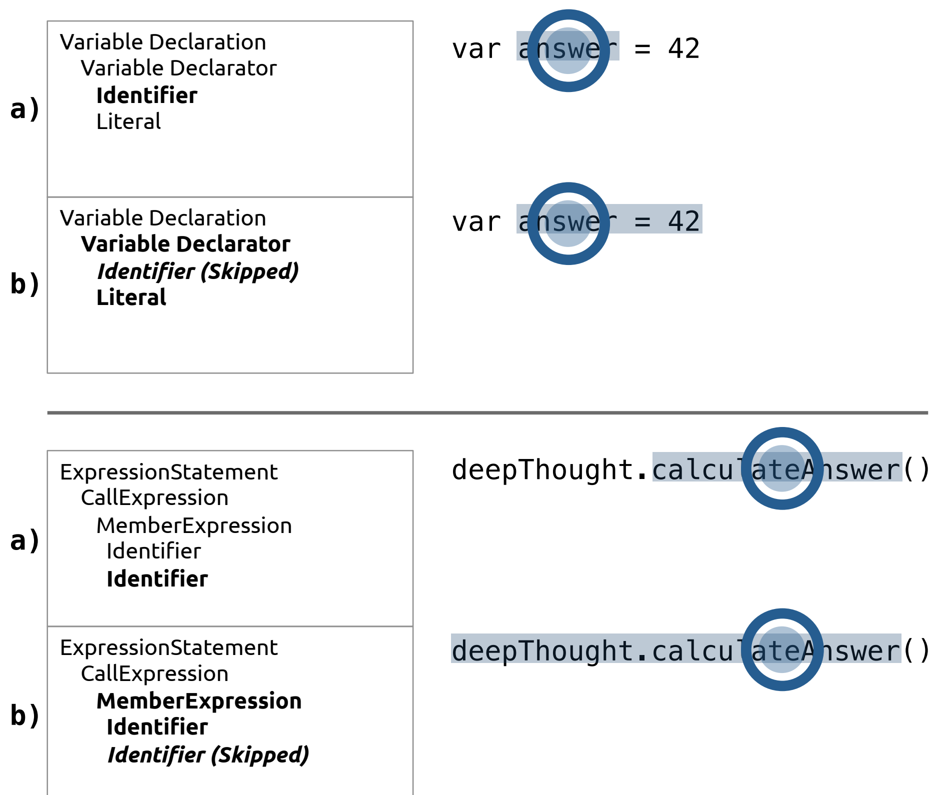


Figure 5.11: The effects of visiting (a) and skipping (b) child nodes. When child nodes are skipped while traversing the syntax tree, the respective parent node will be selected. While this behavior might be desired in the top example, it might be inconvenient in the bottom case.

Here, the recommended method is to select the innermost touched node; that is, to visit all child nodes while traversing the tree. This approach is based on the assumption that users often select *identifier* nodes (which the study results seem to confirm). However, particular node types at higher positions in the hierarchy cannot be directly selected. While touching-and-holding the character "." in the bottom example of Figure

5.11 could be utilized to select the entire *call expression* (including the parentheses), there is no obvious way to select the *expression statement* including a (here optional) terminating semicolon. Semicolons can be selected via adjustment or line selection methods, detailed in the following sections.

Changing the Selected Range

Although the initiation phase quickly selects ranges based on structural boundaries, the user might want to adjust this range further. According to the study results, half of all selections do not precisely align at node boundaries, suggesting that AST selection should not replace the possibility for fine manual adjustments. However, conventional selection handles can be enhanced by supporting both coarse syntactical selection and fine character-wise adjustments. That way, syntax-aware selection continues to be available *after* the initiation phase.

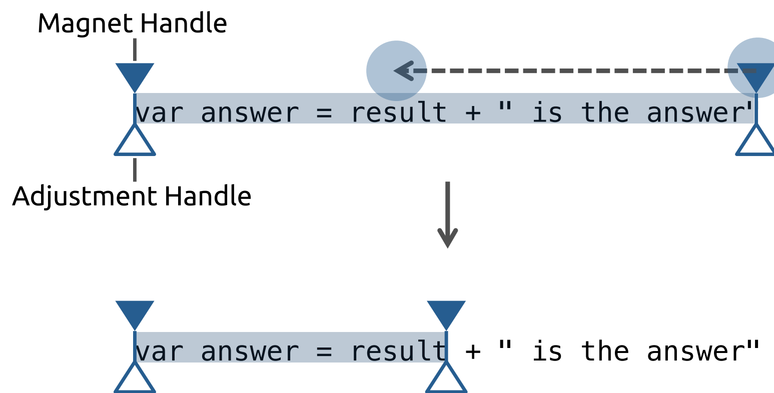


Figure 5.12: Selection handles consisting of a *magnet handle* (top handle) for snapping (limiting) the selection to syntax boundaries of other child nodes, and an *adjustment handle* (bottom handle) for conventional character-wise selection.

Mobile platforms typically render small widgets for adjusting the start and end position of the range by dragging handles to the corresponding character positions. This system can be extended by splitting one handle into two active areas: While the bottom handle works similar to conventional selection widgets, the top handle serves as “selection magnet” and snaps to the next logical syntax boundaries when the user releases the finger. Figure 5.12 illustrates this design.

Since the magnet handles now modify two locations (instead of one as in the initiation phase), the selection mechanism must treat both locations differently: The first handle snaps to the *start* location of its surrounding node, whereas the second handle snaps to the *end* location of its surrounding node. The selection can thus be adjusted to a different range within a child node, or to any descendant, sibling, or ancestor. If the second handle is dragged *before* the character position of the first handle, the logic is reversed (i.e., the second handle is set to the start location of its surrounding node, while the first handle is set to the end location of its surrounding node). As shown in Figure 5.12, this design lets users constrain the selection to a specific child node of the variable declarator (`result`) after the initial coarse selection of the entire assignment. The bottom handles allow for further character-wise adjustment.

Multiple Selections

An increasing number of desktop IDEs have integrated a feature called "multiple selections". This feature allows users to select multiple non-contiguous ranges and modify these ranges at once (e.g., by replacing the contents). The functionality is typically triggered by pressing modifier keys.

The previously presented design lends itself well for implementing the distinction between single and multiple selections based on the touch duration: Repeatedly initiating a touch-and-hold gesture adds a new selection to the previous selection. The most recent selection becomes the *primary selection* that displays the selection handles. A tap gesture clears all active selections.

5.3.2 Selection Gestures and Widgets

In this section, I introduce two gesture-driven selection techniques and one widget-based technique. Like syntax-aware selection, these methods have emerged from the study results. While syntax-aware selection forms the basis, the three approaches presented here either complement the interaction (*selection spans*) or provide enhancements for specific code structures (*selection panning* and *selection rails*).

Selection Spans

Although syntax-aware selection lets users select any part of the code, the approach does not take advantage of *multi-touch* interaction. *Selection spans* allow users to set the selection anchor and selection head through repeated drag-and-hold gestures using one finger or two fingers (Figure 5.13). The interaction for each version works as follows:

- **One finger**: A range is selected by setting the anchor with the first touch-and-hold on the start point. Moving the finger to the end point, followed by dwelling, sets the head. Subsequent dwelling at other end points always adjusts the head, while the anchor remains at the first fixed position. This allows users to readjust the range during the interaction, should they decide to select a different range. Selection handles are displayed when the user lifts the finger.
- **Two fingers**: Although the interaction is performed with both fingers simultaneously, one finger always touches few milliseconds first and thus sets the anchor, whereas the second touch sets the head. In contrast to single-touch interaction, the multi-touch version adjusts both the anchor and the head (or only the head). Selection handles are displayed when the user lifts the finger.

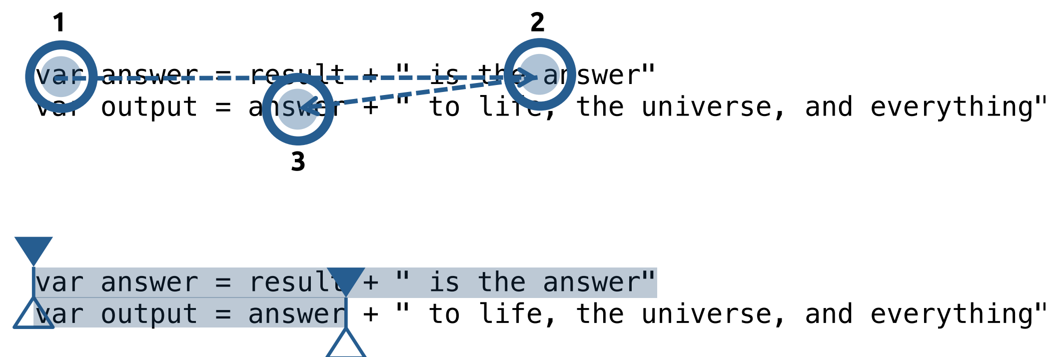


Figure 5.13: *Selection spans*. Top half: One or two fingers set the anchor (first touch-and-hold, no. 1) and head (dwelling, no. 2). While single-touch interaction allows users to readjust the head (dwelling, no. 3), the multi-touch version readjusts both positions during the interaction. Bottom half: Selection handles are displayed when the user lifts the finger(s).

Selections spans allow users to bypass the adjustment of handles by limiting the range already during the *initiation phase*.

Selection Panning

Selection panning uses the gutter area as touch target. The study presented in Chapter 4 has identified this area as popular target for selecting a single line or multiple lines of code. With *selection panning*, touching-and-holding on a line number selects the corresponding line. A line range is selected by dragging towards the end line number and lifting the finger. When the initial touch-and-hold is skipped and the user immediately starts panning on the gutter, only a single line will be selected.

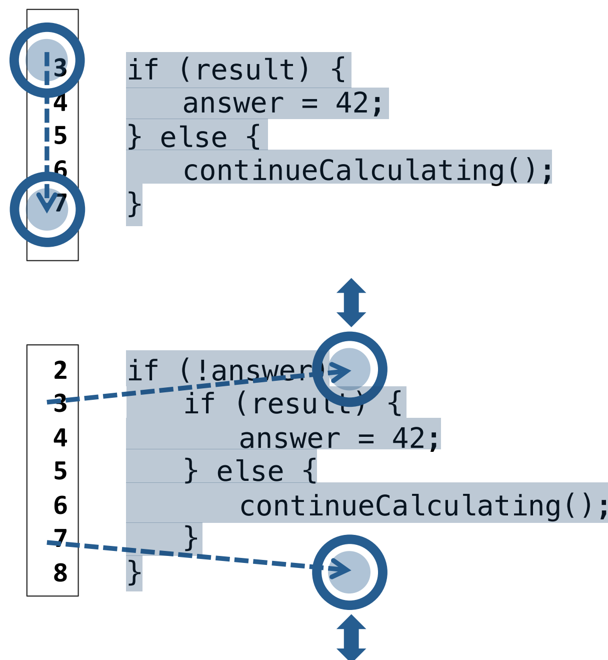


Figure 5.14: *Selection panning*. Top half: Line ranges are selected by panning over the line gutter using single-touch or multi-touch interaction. Bottom half: The touches can be shifted towards the right so that pinching the fingers decreases or increases the range on top of the code.

Alternatively, a two-finger gesture selects a line range with a single operation. When the fingers are lifted, the anchor is positioned at the start of the first touched line while the head is positioned at the end of the second touched line. During the interaction, the range can be readjusted by horizontally sliding the touches towards the right and then pinching the fingers (Figure 5.14). This horizontal shift allows users to increase or decrease the line range on top of the code.

Selection Rails

Selection rails are a widget-based technique for quickly selecting *block statements*, which have shown to be the second most frequently selected node type in the user study. After a touch-and-hold on any part of the line gutter, bold lines are displayed next to *block statements* in the editor area. Tapping on a line (rail) selects the corresponding block. Multiple taps on the same line extend the line range by selecting the parent node of the current block; thus, the selection can be extended up to the root node of the current document.

Figure 5.15 shows five selection rails in a code example containing multiple *block statements*. (*Block statements* also enclose the single-line *consequent* and *alternate* nodes of the *if statement* node.) Invisible hit areas ensure that users do not need to touch the width of the line precisely. A short-tap elsewhere in the editor area hides all selection rails.

```
1   function calculateAnswer() {
2       while (!answer) {
3           if (fetchResult() == 42) {
4               answer = "life, the universe and everything";
5           } else {
6               // nothing
7           }
8       }
9   }
10
11
```

Figure 5.15: *Selection rails* are bold lines that enable quick selection of *block statements*. Multiple taps on the same rail extend the range up to the next parent node. Here, the user has first touched the rail next to the *if statement*. Another touch would select the *while statement*, and so on.

5.4 Conclusion

Selection is an essential and frequently performed operation in code editors. Touch-enabled devices, however, lack interaction methods for executing this operation efficiently. Desktop editors provide a number of keyboard shortcuts for quickly selecting

text elements like words, sentences, and paragraphs. In contrast, widely used mobile platforms have employed UI widgets and limited support of gestures. Since these methods are insufficient for selecting source code, I have proposed touch-centric interaction techniques that take the syntax and structure into account.

In this chapter, I have first introduced the terminology of text selection and described the difference between modeless and modal commands. Examples of the selection concept, as realized by desktop and mobile operating systems, have demonstrated how users typically select text. Furthermore, I have presented a user study that has examined programmers' code selection behavior in a realistic development scenario. While the results have revealed various properties of their performed commands, two particular results have motivated the design of new interaction techniques: First, programmers frequently select code at structural boundaries and second, they select certain node types of the syntax tree much more frequently than other types.

In the final part of this chapter, I have proposed multi-touch interaction techniques for code selection on touchscreens. *Syntax-aware selection* lets users perform quick selection of structural code regions based on AST node boundaries. Additional gesture-driven and widget-based techniques (*selection spans*, *selection panning*, and *selection rails*) provide enhancements for specific code structures. I present implementations of these approaches in Chapter 7.

Opportunities for future work include conducting a follow-up study that investigates the designed methods from a usability perspective. This might also show if the programmers' expected selections match those generated by the devised interactions.

Chapter 6

Creating Source Code

While the previous two chapters have addressed the interaction with *existing* source code, this chapter concentrates on the interaction of creating *new* code. Researchers have long sought methods for efficient text entry on touchscreens. Entry of source code adds new challenges, primarily arising from the need for code-centric methods to match the well-established and keyboard-driven techniques of their desktop counterparts.

This chapter is divided into five main sections: In the first section, I introduce the challenges of touch-centric code input. In the second section, I describe the state-of-the-art of standard code creation mechanisms in desktop editors. Following that, I present the model and design approach of a custom keyboard for code entry. In the fourth section, I report the results of a user study centered around entering code with this custom keyboard. The final section proposes improvements based on user feedback and presents simulations of an enhanced model for code entry.

6.1 Challenges

So far, I have outlined strategies for editing existing source code. However, typing new code into a system that lacks a physical keyboard remains a major issue. Although mobile platforms typically provide on-screen (virtual) keyboards for typing characters into applications, a number of problems are associated with their use.

6.1.1 Fat Fingers

Haptics are perhaps the most noticeable difference between on-screen keyboards and physical keyboards. Missing tactile feedback coupled with inaccuracies between *intended* and *actual* touch locations (known as the *touch offset*) worsens the user experience of typing on flat surfaces. One of the most well-known interaction issues with any touch-based system is the “fat finger problem”: The combination of the user’s entire fingertip being a potential contact point and the finger occluding the touch target impede precise pointing at the intended location. Although this random noise contributes to touch inaccuracies, researchers have argued that up to 70% of inaccurate touches do *not* arise from “fat fingers” but from a systematic error that depends on the user and the finger posture [HB10]. Hence, systems could more accurately recognize touches if they adapted to users and sensed certain properties such as the angle at which the finger touches the surface.

6.1.2 Touch Model

In order to improve the accuracy, research projects have incorporated information about the users’ hand postures (i.e., which fingers of which hand are used) [GJM+13]. Similar to the previous approach, applications should learn the users’ intended touch locations in relation to the actual touch locations and adapt over time. Although user adaptation has been shown to improve the touch accuracy, a known problem with machine learning techniques is that they require collected training data until they become useful. Furthermore, most available mobile devices do not provide information about the user’s grip, finger posture, fingertip, or the size and angle of the touched area. Application developers are limited to only working with basic data about the locations of individual touches. These constraints have stimulated research on statistical methods that generate substantial improvements by considering only touch locations and potential target objects in the UI [BZ13].

Besides user adaptation and probabilistic methods, researchers have proposed a variety of input methods ranging from magnifying touch areas, shifting touch targets, and other interaction techniques, some of which I have mentioned in Chapter 3. Most of the mentioned strategies have attempted to improve the *touch model* of text entry.

6.1.3 Language Model

The second important component for efficient text entry is the *language model*. Spelling correction and completion, for example, have become standard on mobile platforms. When users inadvertently hit the wrong key, the system displays a list of corrections; when users type only part of a word, the system displays possible completions. More advanced approaches such as Apple's *Quick Type*[1] additionally analyze the user's context to improve the quality of suggestions. Moreover, research has shown that both *correction* and *completion* are not mutually exclusive but rather supplement each other [BOZ14]. Both capabilities can be optimized without losing the major advantages of each objective. This optimization problem, however, is not trivial: The keyboard of the Android platform, for example, is driven by 21 weighted parameters affecting the quality of the text entry system [BOZ14].

6.1.4 Text vs. Source Code

Overall, the best results are probably achieved when the *touch model* and the *language model* are combined [WPR$^+$14]. However, existing solutions only cover conventional scenarios such as messaging or writing text documents. Programming differs from the typical use cases for text entry methods. It might even amplify known problem areas for the following reasons:

- Typing the text elements of source code may be more onerous than typing regular textual content.
- Various special characters are disproportionately often accessed.
- Auto-completion is not only a convenient but rather essential and frequently performed operation.
- Conventional spelling correction systems are not optimized for programming languages.
- Programmers interleave typing with performing keyboard shortcuts.

As a consequence, the touch model and language model must either be optimized for source code and/or be complemented by a third major building block. To make it

[1] `https://www.apple.com/de/ios/ios8/quicktype/`

clear which properties should compose a code-centric text entry component, I present existing code creation mechanisms in the following sections.

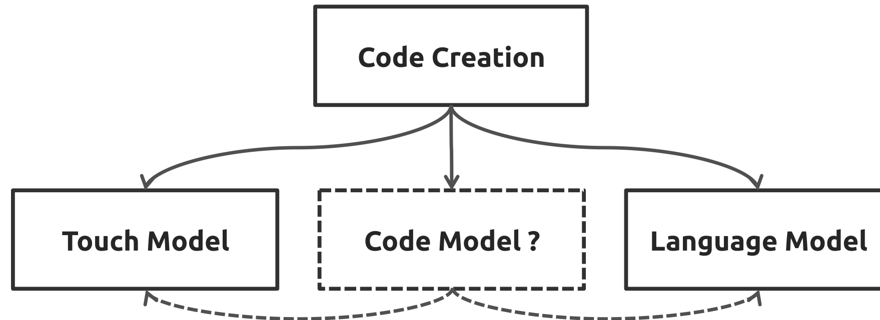


Figure 6.1: Touch-based text entry systems are typically driven by an underlying touch model and language model. A third building block might need to be added for optimized code entry.

6.2 Code Creation in Desktop IDEs

Obvious methods for generating code include manually typing in the editor or pasting a snippet from an external source. This section lists additional ways of creating code through IDE features, which I have categorized into five main areas: Smart Typing, Code Completion, Code Hints, Code Templates, and Code Generation.

Although modern desktop IDEs differ in their naming and implementation details of code creation methods, the concepts are similar between applications. *Code Assistance* is perhaps a suitable umbrella term for these collections of functionality that support programmers in writing source code more quickly and with fewer errors. On the one hand, users of touch-enabled editors would certainly expect similar functionality; this expectation has also been confirmed by the user study presented later. On the other hand, adopting the user experience from desktop applications appears inadequate. Therefore, code creation requires–similar to other functional areas–changes pertaining to the different interaction paradigm on touchscreens.

Rather than showing the code creation features of every major IDE, I clarify each principle using the example of the popular Eclipse IDE.

6.2.1 Smart Typing

Smart Typing refers to the capability of IDEs to insert pairs of specific characters such as parentheses, brackets, quotation marks, or comment signs *during* typing. When the programmer inserts the first character of a pair, the IDE automatically inserts the matching closing character. For example, entering an opening brace automatically inserts the closing brace, whereupon the programmer continues typing the code between the braces. *Smart Typing* also applies to indentations: Pressing enter not only creates a new line but also indents the line accordingly. Overall, this feature helps programmers saving keystrokes when entering special characters.

Figure 6.2: *Smart Typing* automatically inserts matching character pairs during typing. The cursor is placed so that writing continues between the character pair.

6.2.2 Code Completion

Code Completion (sometimes also called *Auto-complete*) is one of the primary ways for accelerated typing of code. When programmers trigger this feature, the editor *completes* partially typed code. A typical use case is the completion of member functions and attributes: After typing the name of an entity, followed by a *trigger character* such as “.”, the programmer selects an entry from the displayed list of suggestions. The list of choices is then filtered with each subsequent keystroke. Accepting the selection by pressing *Enter* (sometimes also *Tab*) inserts the completion and hides the pop-up list. A keyboard shortcut usually lets programmers manually show completions at arbitrary locations in the source code.

In many IDEs, code completion does not stop with completing members but also includes keywords, variable names, parameters, missing parentheses, and even paths to local files. More advanced code completion engines attempt to consider the current context and pre-filter the list of suggestions. The NetBeans IDE, for example, displays *smart suggestions* in a separate section at the top of the list. *Smart suggestions* are choices that might be more relevant to the user’s current context. Since the list can contain a considerable number of suggestions, this behavior accelerates the insertion of more likely completions.

IDEs like IntelliJ or Visual Studio optionally perform *instant completion*, that is, almost every keystroke (rather than only a special character) triggers completions. Because the constant display and updating can be distracting, some IDEs show the pop-up only after a delay and/or only at certain characters. Most IDEs provide settings for configuring trigger characters, delays for pop-up lists, and the sorting of suggestions.

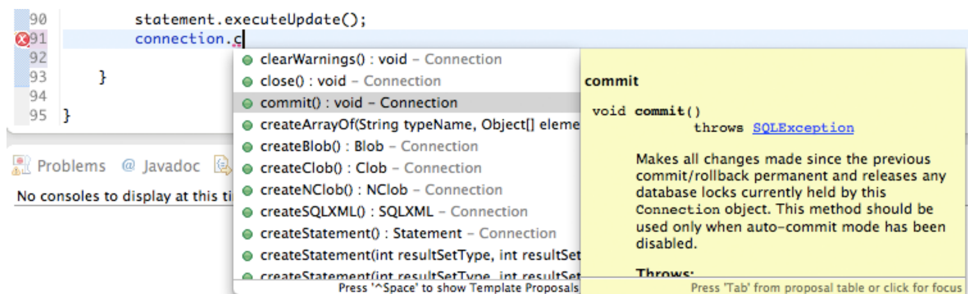


Figure 6.3: *Code Completion* displays a filterable list of suggestions that the programmer can accept and insert.

6.2.3 Code Hints

Code Hints provide corrections for potential errors. Programmers also repurpose this feature to create code by intentionally introducing errors and thereby invoking the display of a hint. When the editor detects errors, it displays hints as small widgets (e.g., a light bulb icon) in the gutter area or directly above or below the code as inline-widget. Clicking on the widget or executing a keyboard shortcut then shows a list with possible corrections for the erroneous code.

When the programmer selects an option, the IDE tries to repair the code. The action may also result in code being generated: The programmer could intentionally refer to code that has yet to be written, for example by calling an unknown method. The IDE detects the unresolvable method name and shows the appropriate code hint that allows the programmer to create the method definition. The same process could be used for generating variable declarations or other code structures.

Eclipse calls *Code Hints* “Quick Fixes” and supplements the feature with “Quick Assist”. Programmers can use the latter to generate code even when no error exists, for instance for performing local code transformations. Some transformations, such as adding an *else block* to an *if statement* or extracting a local variable, create new code. However,

the large number of available code hints begs the question to what extent programmers really exploit these features for code generation purposes.

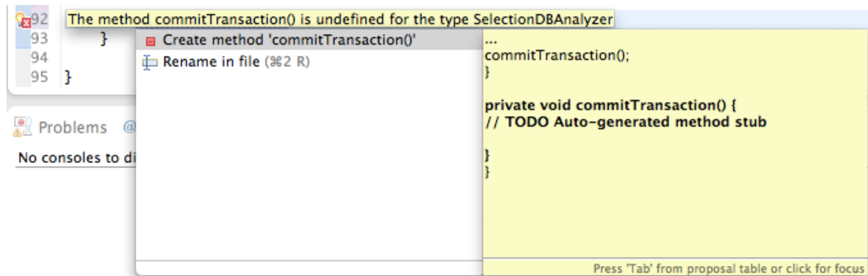


Figure 6.4: In addition to repairing erroneous code, code hints can generate new code when the editor cannot resolve a method that the programmer has attempted to call.

6.2.4 Code Templates

Code Templates (sometimes called *Live Templates*) allow programmers to quickly insert blocks of code and only fill in the required placeholders of the template. The templates are usually triggered by typing a configured abbreviation, followed by using either the code completion mechanism or a reserved keyboard shortcut. After expansion of the template, the editor marks the placeholder regions of the inserted code. To create a new private method, for instance, the programmer types "private", inserts the suggested template and then navigates through the placeholders to modify the return type, parameters, and method body. Some editors enforce a valid context before a template can be inserted.

Most IDEs let users configure custom templates. Applying this mechanism, programmers do not need to recall the exact pattern of a rarely used structure, or, perhaps more commonly, they quickly insert and modify frequently used code snippets.

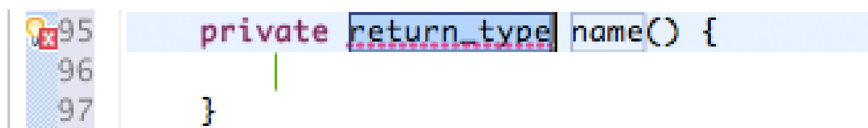


Figure 6.5: An active code template for defining a new method. Using the *Tab* key, the programmer can navigate back and forth between the editable regions and fill in the details. Pressing *Esc* ends the template, pressing *Enter* moves the cursor to the location displaying the cursor marker.

6.2.5 Code Generation

Here, *Code Generation* refers both to dialog-based generation methods and to all other mechanisms not fitting into the previous categories. Some IDEs, for instance, provide dialogs for generating the *accessor methods* of selected fields in a class. Other features generate overriding methods in subclasses or surround selected code with particular statements (e.g., *if* or *for*).

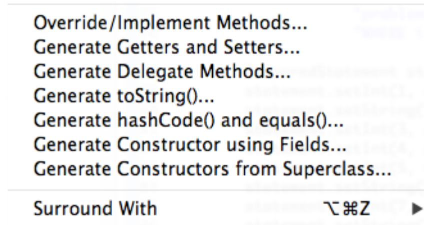


Figure 6.6: Menu options for code generation in the Eclipse IDE. Options starting with "Override" or "Generate" lead to dialogs containing adjustment parameters for new code to be generated.

6.3 Towards a Code Entry Keyboard

In this section, I present the design of a custom keyboard that has been optimized for code entry. The design has been evaluated in a user study and is part of the implementation shown in Chapter 7.

6.3.1 General Design Approach

Although the design of the keyboard is not tied to a particular platform, the concrete implementation used in the study has been developed for iOS and the iPad. To make the differences clearer, I briefly introduce the default keyboard of this platform and highlight its shortcomings concerning code entry. The following sections refer to the iPad default keyboard as *IDK* and the code entry keyboard as *CEK*.

Figure 6.7 shows the native iOS (version 7.1.2) keyboard. In landscape mode on an iPad Air (screen size: 19.7 cm x 14.8 cm), the on-screen representation takes up 100% of the horizontal screen space and about 46% (~6.8 cm) of the total vertical screen

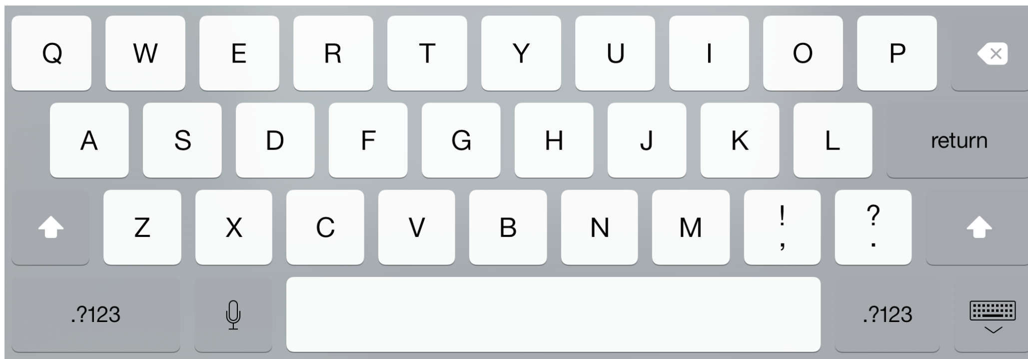


Figure 6.7: Default English QWERTY keyboard of the iPad in landscape orientation.

space. The rectangular area of the standard letter keys is 1.52 cm x 1.44 cm, the horizontal gap between the keys is 2.7 mm, and the vertical gap is 2.5 mm.

The English *QWERTY* layout, by default, shows all letter keys, *Shift* keys for typing uppercase characters, special keys for toggling the entire layout to show numbers and special characters, and other standard keys such as *Backspace* and *Enter*. The functionality of *Enter* depends on the active application (e.g., the label changes to *Go* when typing into the address bar of the built-in web browser). While holding *Shift* enables typing of uppercase letters, a single tap only enables uppercase mode for the following letter and then toggles back to lowercase mode.

Special characters are distributed over two separate layouts. The second layout can be accessed by pressing a special button that replaces *Shift* in the first layout. For example, to access "=", the user has to first press the button labeled ".?123", then "#+=", and finally "=". Keys are triggered only after the finger is lifted from the surface. Touching-and-holding certain (visually indistinguishable) keys shows a small temporary pop-up displaying related items above the touched area. A special key in the lower right corner allows users to *undock* the keyboard, that is, to freely move the keyboard from the bottom to another vertical position. Furthermore, the keyboard can be split into two smaller halves by performing a pinch gesture over the keyboard. Changing to another localization of the keyboard can result in changed dimensions of the keys. For example, changing from the English to the German layout introduces keys for Umlauts ("äöü") and "ß"; as a consequence, the width of all keys is reduced accordingly.

As far as code entry is concerned, the described standard keyboard has a number of shortcomings:

- The keyboard covers almost half of the vertical screen space.
- Special characters that are frequently used for coding are onerous to access.
- There is no support for accelerated triggering of text editing commands (e.g., moving the cursor to the end of the line).
- The keyboard does not exploit multi-touch and gestural interaction.

An optimized code entry keyboard might need to satisfy the following requirements in order to address these issues:

- Users should be in control of the keyboard size.
- Special characters for coding should be quick to access.
- Standard text and code editing commands should be quick to perform.
- Multi-touch and gestural interaction should enhance the user experience.

Furthermore, as emphasized in [FWW11], virtual keyboards could benefit from customization. In contrast to physical keyboards, software-based keyboards could adapt to the user or let the user configure specific elements of the layout. (The latter might be attractive for programmers who appear to be particularly receptive regarding customizations of their code editing environment.)

6.3.2 Keyboard Layout and Size

Figure 6.8 shows a screenshot of the initial layout for the proposed code entry keyboard. Compared with the IDK, the CEK has a reduced height (~5.5 cm vs. ~6.8 cm) and displays an additional key per row. The entire set of keys is horizontally and vertically centered within the available lower area of the keyboard. All keys maintain a horizontal and vertical gap of 1.9 mm; the size of individual keys is adjusted accordingly. By default, the keys are thus smaller than those of the IDK, but users can progressively increase the size. The top part of the keyboard, taking up 7.8 mm of the 5.5 cm, is reserved for supportive widgets.

Users may choose to resize the keyboard by performing a standard pinch gesture over the keyboard. Pinching resizes the bottom part of the keyboard proportionally while the top bar maintains its vertical height. The resized area can extend over the screen edges, clipping the contents if necessary. Employing this resizing option, users can

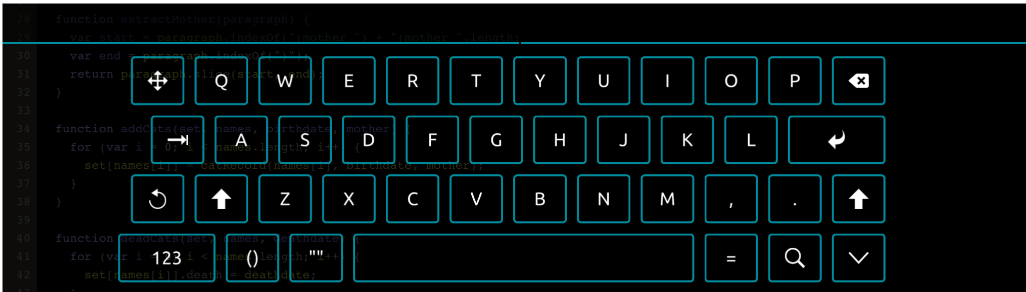


Figure 6.8: Initial layout of the proposed keyboard for code entry.

intuitively adjust the key sizes to their liking. Moreover, the keyboard can be undocked and freely dragged to any location on the screen, both horizontally and vertically. Dragging is initiated either after touching-and-holding on the top bar or by dragging simultaneously with pinching (Figure 6.9).

Whereas the keyboard “floats” above the editor contents in the undocked state, the docked state positions the keyboard at the bottom of the screen and shifts the editor content to the top (i.e., the editor content does not cover the area below the keyboard). When the keyboard is set to the docked state, it keeps the size that was configured in the undocked state. Resizing and repositioning is disabled when the keyboard is docked.

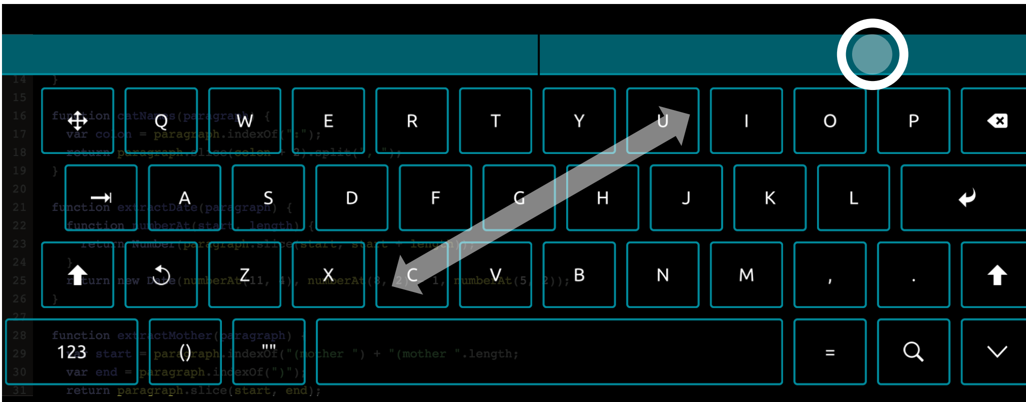


Figure 6.9: In the undocked state, the keyboard can be resized using a pinch gesture (double-headed arrow) and dragged to new locations after performing a touch-and-hold gesture (circle).

Similar to the IDK, the CEK uses the standard *QWERTY* layout but replaces some IDK keys and adds new keys that are more adequate for code editing. All keys provide a configurable default behavior when pressed. A gesture-driven menu system (described

after this section) triggers additional behaviors when activated. The following keys differ from the IDK:

Cursor (row 1, key 1)
Moves the cursor similar to the mechanics of the arrow keys and modifier keys on physical keyboards: The cursor can be moved one character to the left or right, one line up or down, to the start or end of a line, and one word to the left or right.

Backspace (row 1, key 12)
In addition to the default backspace functionality, the key enables deleting up to the start or end of the word before of after the cursor, deleting the whole line, and deleting the line before or after the cursor.

Tab (row 2, key 1)
Inserts a *soft tab* (i.e., the number of spaces corresponding to the width of a tab character at the current cursor position). The key also indents or outdents a line/selection by one unit, or automatically indents the line/selection.

Enter (row 2, key 11)
Applies automatic indentation after inserting a new line.

Undo/Redo (row 3, key 1)
Performs an undo/redo operation for the last change.

Shift (row 3, key 2 and row 3, key 12)
In contrast to the toggling *Shift* key of the IDK, this key initiates a *quasi-mode*, that is, it enables uppercase letters only as long as the finger touches the key and disables the mode when the finger is lifted.

Comma (row 3, key 10)
Inserts a comma, semicolon, underscore, hash sign, and other special characters.

Period (row 3, key 11)
Inserts a period, exclamation mark, question mark, colon, and other special characters.

Special (row 4, key 1)
Replaces the current layout with a new layout displaying all available special characters in a single layout (see Figure 6.10).

Parentheses (row 4, key 2)
: Inserts parentheses, square brackets, angle brackets, braces, and other special characters. When character pairs are inserted, the cursor is positioned between the characters.

Quotes (row 4, key 3)
: Inserts double quotes, single quotes, and other special characters. When character pairs are inserted, the cursor is positioned between the characters.

Equals (row 4, key 5)
: Inserts an equal sign and special characters for arithmetic operations.

Search (row 4, key 6)
: Triggers the editor's search functionality and shows the search/replace widget. (Pressing this key had no effect in the user study.)

Undock/Dock (row 4, key 7) Toggles docking and undocking of the keyboard. In the undocked state, the keyboard can be resized and repositioned while "floating" above the editor content.

Overall, this design attempts to stay as close as possible to the familiar *QWERTY* layout but adds frequently needed keys for code entry. While the *Special* layout displays all special characters in *one* separate layout (Figure 6.10), all characters are also accessible from the initial layout through gesture-driven interaction. In addition, the layout and key behaviors are fully customizable via a JSON configuration file. A visual editor for this configuration file could provide customization options for users who prefer different arrangements.

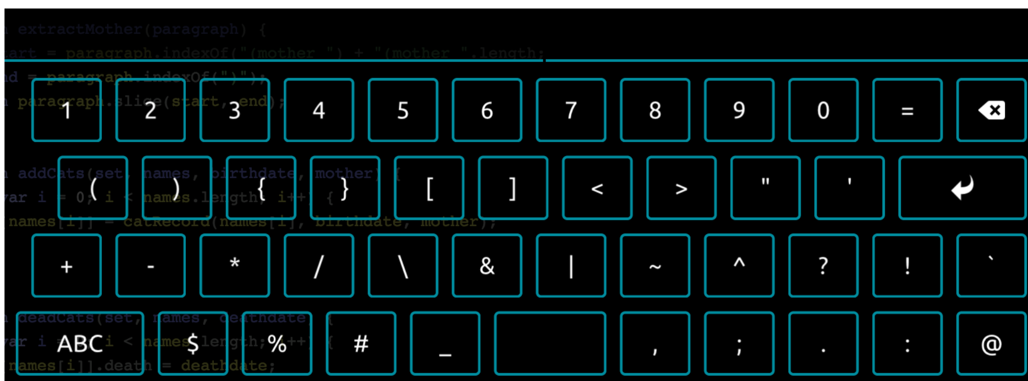


Figure 6.10: Although all relevant special characters can be accessed from the basic layout, an additional layout displays all special characters at once.

6.3.3 Gestures and Marking Buttons

In the previous section, I have mentioned additional actions triggered through gesture-driven interaction. The CEK executes the default functionality for a key when the finger is lifted from the surface. This behavior is in line with the IDK but differs from physical keyboards: Physical keyboards afford both a *touched state* and a *pressed state* for each key; the latter triggers the key, whereas the former lets users rest the fingers on the keyboard. Since virtual keyboards do not provide a *pressed state*, the key is usually triggered as soon as the user lifts the finger (*released state*).

In the CEK, the *touched state* displays a *marking menu* [Kur93] (Figure 6.11) after a touch-and-hold gesture. (Marking menus are described in Chapter 3.) The key thus acts as activation area for the menu. When the user then drags the finger from the center to an item placed at one of the eight directions, the item is highlighted. Lifting the finger selects the item and hides the menu; lifting the finger over the center cancels the menu selection. Exploiting expert-mode of marking menus, users can rapidly select an item by flicking the finger from the key towards one of the eight directions. Expert-mode skips both the initial delay and the visual representation of the menu and thereby enables fast access to individual entries during typing.

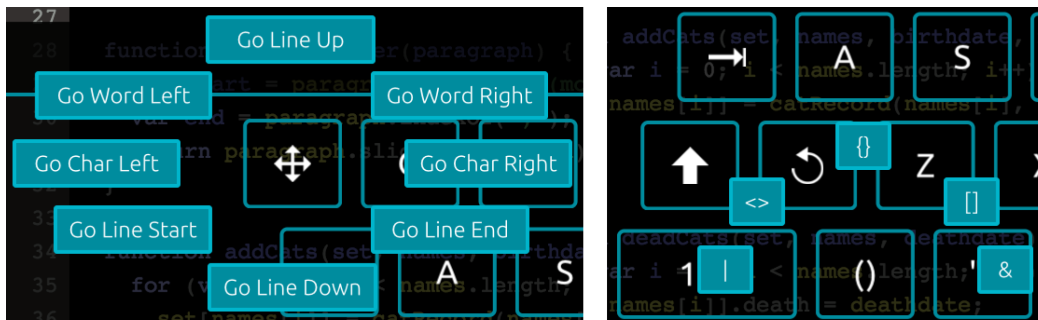


Figure 6.11: Marking menus for the *Cursor* key (left) and the *Parentheses* key (right). Marking menus are displayed after a touch-and-hold gesture on a key and show up to eight additional actions (novice-mode). All actions can be rapidly executed by performing directional flick gestures (expert-mode).

With non-hierarchical marking menus, each key supports up to nine different actions (the default action plus eight menu items). When the keyboard is docked at the bottom of the screen, the lower three menu directions extend over the bottom screen edge in novice-mode. Custom configuration could, therefore, leave the clipped items at screen edges unassigned while expert-mode would still allow for gestural access.

It is perhaps most reasonable to group related keys or actions around the base key (e.g., grouping square brackets, angle brackets, and braces around parentheses). However, since the direction of a menu item affects the resulting gesture, items should not be placed at different directions arbitrarily. A common practice with marking menus is assigning an action and the inverse of the same action to opposite directions. This arrangement, although not always possible, might more naturally reflect the users' way of thinking. For example, *undo* and *redo* or *indent* and *outdent* are appropriate action pairs for opposite directions. Another strategy could be placing items at those directions that are easier or faster to access with gestural movements. "Off-axis" items (i.e., the four corner items at diagonal directions) are more difficult to access than horizontal and vertical "on-axis" items [KB93], and vertical movements are faster than horizontal movements [Sam13]. These findings could be combined with data about the character frequencies of a programming language so that frequently performed characters are assigned to the vertical and horizontal directions.

Due to the touch-and-hold gesture for menu activation, resting fingers on the keyboard may inadvertently trigger the menu display. This ambiguity could be resolved by recognizing the number of fingers simultaneously touching the surface: The menu display could be disabled when a certain number of fingers is resting on the surface or as soon as "finger resting patterns" are detected. (The keyboard version used in the study, however, does not implement this behavior.)

The approach of integrating marking menus is similar to the built-in pop-up menus of the IDK. The IDK uses such menus to display extra characters. On physical keyboards, in contrast, special characters or alternate functionality is triggered via modifier keys. Marking menus additionally exploit gestural capabilities and reduce the needed visual attention through their fast eyes-free expert-mode. The next section shows how this interaction technique has been utilized for triggering code templates.

6.3.4 Code Templates

Code templates are pre-defined code snippets, which programmers typically insert by pressing a keyboard shortcut, followed by navigating to placeholders and filling in code. Template functionality becomes particularly useful in touch-based editors since it can save a considerable amount of keystrokes. To let users invoke code templates during typing, the CEK applies the previously mentioned interaction to letter keys: A marking menu provides gestural access to all code templates starting with the same

letter of the key. For example, the letter key "F" inserts templates such as *Function*, *for*, or *for...in* (Figure 6.12).

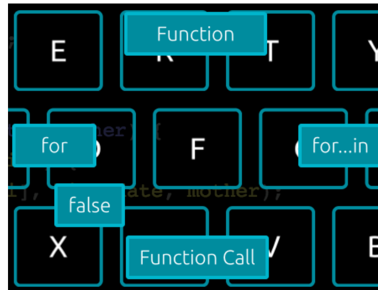


Figure 6.12: Code templates are mapped to keys matching the first letter of the template. Marking menus are employed for invoking the templates.

This design shows some similarity to *Augmented Letters* [RMG+13] since it associates the first letter with marking menu items. *Augmented Letters*, however, are different in that they are triggered by drawn shapes instead of permanently displayed keyboard keys (e.g., the user draws an "S" shape to invoke a marking menu containing all items starting with "S"). The technique presented here is similar in that it takes advantage of linguistic mappings, which may help users learning and recalling menu items [RMG+13]. Instead of having to learn full abbreviations, such as those of desktop IDEs, users only need to remember the first letter and–for accelerated execution–internalize the direction of the menu item.

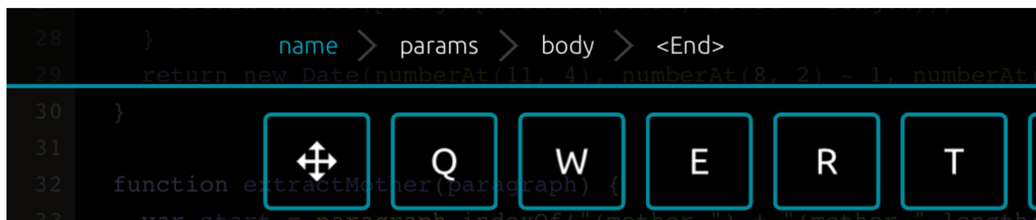


Figure 6.13: After insertion of the code template, the keyboard renders a supportive widget for navigating between editable regions. In this case, the programmer has inserted a *Function* template and can now navigate between the name, parameters, and body of the function.

Besides enabling template insertion through mnemonic marking menus, the keyboard renders a supportive widget for template navigation (Figure 6.13). The top left area of the keyboard displays all placeholders in an ordered horizontal list. By tapping on the items, the user can directly jump back and forth between the marked regions of the template and continue modifying the selected code. The widget is activated as soon

as the template is inserted, and the displayed placeholders depend on the template configuration that is loaded from an external file. The last item in the list (“<End>”) lets users exit the template at any point.

6.3.5 Code Completion

Programmers frequently use *code completion* in IDEs [MKF06]. The suggestions are usually displayed in pop-up menus when the user types certain characters or manually invokes the menu with a keyboard shortcut. Instead of rendering the list of completions within the source code, code completion as proposed here, is integrated into the keyboard: The CEK renders suggestions as horizontal list in the top widget area of the keyboard (Figure 6.14). The left half already displays navigational items for active code templates; the right half still provides sufficient space for the list of suggestions. The list is scrollable through swipe motions, although scrolling should be rarely required since suggestions are filtered with each keystroke. A completion is applied when the user taps on an entry. The direct selection of completions could be regarded as advantage over desktop IDEs where users have to perform additional keystrokes when they wish to apply an entry other than the topmost item.

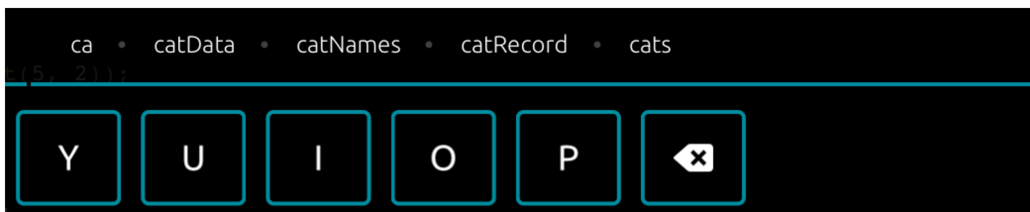


Figure 6.14: Widget for code completion in the top right area of the keyboard. In this case, the programmer has started typing the prefix “ca” and is instantly provided with a list of selectable suggestions from the code completion engine.

Some IDEs refresh the list of completions after each keystroke. This behavior, however, can be distracting because it continuously updates the pop-up menu at the cursor position. In the CEK, completions are “always on” and updated instantly, but their visual presentation may be less distracting. The placement of suggestions within the top area of the keyboard is less obtrusive since it does not interfere with the cursor. Furthermore, the inline-menus of desktop IDEs might be hard to use on small touchscreens. Here, the distance to reach each suggestion while typing remains near constant, and each item provides a sufficiently large hit-area for finger touches.

6.3.6 Underlying Models

This section outlines how the three building blocks, as illustrated in Figure 6.1, drive the CEK. (Improvements to this model are proposed at the end of this chapter.) The models were intentionally kept simple for the CEK version used in the study. Since the CEK already introduces both a modified keyboard layout and new UI components that should be evaluated, the effects of model variations would have been hard to isolate.

The touch model does not use any offset modeling or adaptation techniques. A keystroke is detected by checking if a touch falls within the visual boundary of the key area and by distinguishing between keystrokes and menu items (novice- and expert-mode). Four main parameters are used to determine which action should be recognized:

AllowableKeyMovement
: The action is a potential keystroke if the distance between the first touch and subsequent touch events of the same finger lies within a certain threshold.

MinimumFlickDistance
: The action is a potential flick gesture if the distance between the first touch and the last touch of the same finger covers at least the allowable key movement (`AllowableKeyMovement`).

MinimumFlickVelocity
: The action is a potential flick gesture if the first touch at least reaches a certain velocity threshold on subsequent touch events of the same finger.

MinimumPressDuration
: The action is a potential menu invocation if the first touch is held still at least for a certain duration threshold.

When the system recognizes a pinch gesture over the keys, existing touch events are canceled in order to prevent unwanted keystrokes when resizing the keyboard. Also, the CEK does not have limits regarding the number of simultaneous touches.

As far as the language model is concerned, the CEK in its implementation for the study does not use spelling correction or predictive features. A basic language model is realized through a code completion engine[2]. Suggestions are instantly updated

[2] `http://ternjs.net/`

after each keystroke and inserted as previously described. The engine only uses *prefix matching* but does not apply *CamelCase matching* or *subsequence matching*. For instance, typing “ca” matches “catNames” but typing “cN” or “nam” does not.

The third building block of Figure 6.1, the “code model”, extends the touch model and language model with code-related features. In the proposed design, this extension is realized through features such as:

- A keyboard layout optimized for entering source code tokens.
- Gesture-driven invocation of special characters and smart typing.
- Gesture-driven invocation of editing commands and code templates.
- Supportive UI widgets for template navigation and code completion.

Code Hints (in Eclipse terminology: *Quick Fixes* and *Quick Assist*) are not yet supported. While most code-related enhancements are currently realized through UI techniques, the term “code model” also implies added intelligence, such as modeled touch offsets or predictive features while typing code. These enhancements are addressed later in this chapter.

6.4 User Study

The purpose of the user study is to evaluate the suitability of the CEK in a code entry task, uncover potentials usability flaws, and quantitatively measure text entry properties. Hence, the analysis includes both quantitative and qualitative results, both of which are basis for a proposed design revision given at the end of this part.

6.4.1 Participants

10 participants (9 male, 1 female) volunteered for the study and filled in a questionnaire (Appendix C) before the test. They were asked to specify their programming experience, frequently used applications for software development, and their usage of devices with touchscreens.

Ages ranged from 22 to 26 years ($M = 24.2$). All participants were undergraduate or postgraduate students of media informatics or information science. While two students

had only between 1 and 2 years of programming experience, most students indicated between 3 and 4 years or more than 5 years.

4 participants reported that they use smartphones with touchscreens “frequently”, 6 selected “always”. As for tablet usage, 1 selected “never”, 6 “rarely”, and 3 “frequently”. 9 participants were right-handed, 1 was left-handed.

When asked for frequently used IDEs and text editors, the following applications were named (number of mentions in parentheses): Eclipse (10), Sublime Text (7), Notepad++ (4), Visual Studio (2). As frequently used languages for programming or scripting, participants mentioned: Java (10), JavaScript (9), PHP (4), HTML (4), C# (3), Python (3), XML (2).

6.4.2 Test Setup

The task was performed on an iPad Air tablet with a screen resolution of 2048 x 1536 pixels (264 pixels/inch). The test application displayed a full-screen editor that used the CEK as default input method. In addition, the editor displayed a line gutter on the left side, a custom-designed cursor, and buttons for adjusting the font size. The system logged all interaction events for later analysis into a local SQLite database on the device. Figure 6.15 shows the screen with an opened demo task.

6.4.3 Tasks and Procedure

First, participants were given instructions on how to use the system. Specifically, they were introduced to the following features of the test system:

- Cursor (dragging the cursor to a new location; double-tapping the cursor to toggle between showing and hiding the keyboard).
- Special Keys (*Cursor, Tab, Undo, Shift, Special Characters, Parentheses, Quotes, Equals*).
- Docking/Undocking (undocking, moving, and resizing the keyboard; docking the keyboard using the adjusted size; orienting the editor to portrait mode or landscape mode).
- Marking menus (both novice-mode and expert-mode).

- Code Templates (inserting templates and navigating between placeholders, demonstrated by example templates for *var*, *function*, *for*, *if*, and *return*).
- Code Completion (inserting suggestions using the widget).

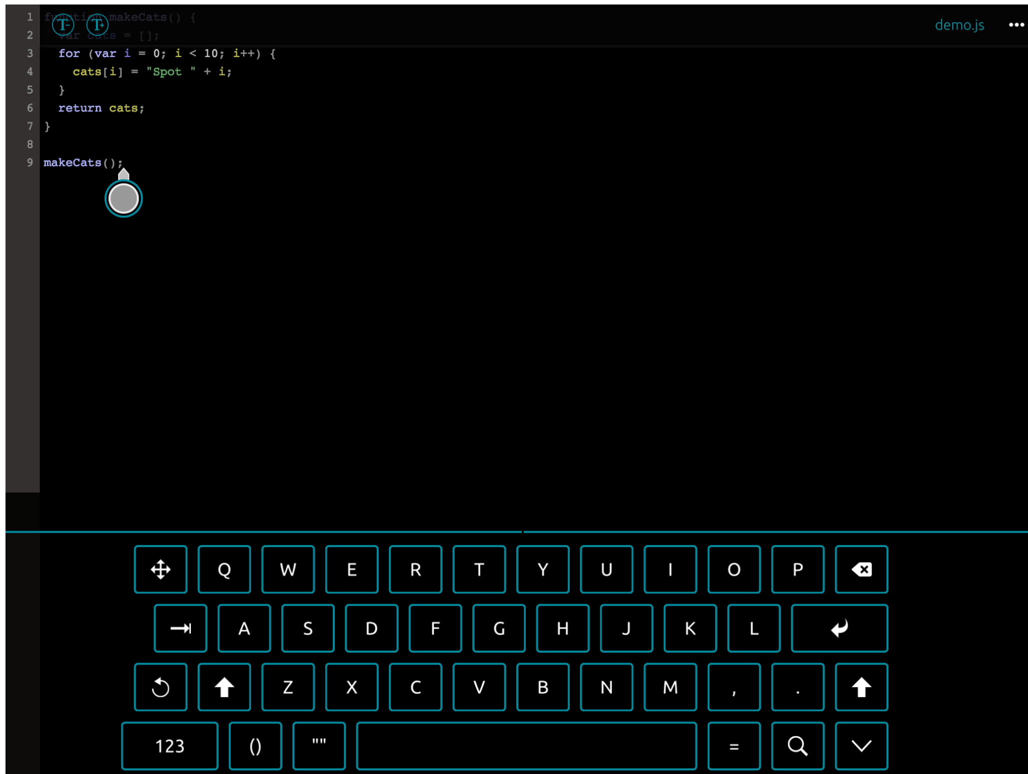


Figure 6.15: Screen after opening the demo task of the CEK study. The CEK is docked at the bottom of the screen. A custom-designed cursor is displayed at the end of the code example. Two buttons for increasing or decreasing the font size are displayed in the bar at the top of the screen. The bar could be hidden by pressing the icon next to the file name on the right edge.

After the instructions, the participants could familiarize themselves with the system until they felt confident in using the keyboard. This phase typically only took about five minutes on average. Furthermore, they were asked to configure the editing environment to their liking, including the tablet orientation, editor font size, keyboard position, and keyboard size. To enforce at least a minimum of typing before the task, the participants had to enter the 9 lines of code of the demo task.

After the introductory phase, a program from a printed A4 sheet had to be transcribed as fast and accurately as possible. Since the total task duration had been regarded as

too long in a pilot test, the program was stripped to about 60% of its original length. The final program (Appendix C) contained 50 lines of code (1012 characters), written in JavaScript. The source code was coherent and contained typical programming constructs such as *function definitions*, *variable declaration*, *for statements*, *if statements*, *array* operations, and *string* operations. Although the participants could correct errors, they were instructed to find a reasonable compromise between speed and accuracy. (Text entry was not constrained.)

6.4.4 Results

The reported results in this section include various measures of text entry performance. The metrics are based on the *method-agnostic* measures (WPM, KSPC, Error Rate) and *method-specific* measures (Selection Deviation) given in [Mac07]. Each section lists the corresponding formula to make it clear how each value has been calculated. According to the terminology given in [Mac07], the program that participants had to type is referred to as the *presented string* (P); the final, resulting string entered by each participant is referred to as the *transcribed string* (T); and the series of the participant's performed actions (keystrokes and editing commands) as the *input stream* (IS). Also, the results include the keyboard configurations and the frequencies or properties of: keys, gestures and menu selections, code templates, and code completions. The section concludes with qualitative feedback and recorded observations.

Task Duration and Typing Speed

Participants took, on average, 15.1 minutes ($M = 15.1, SD = 2.4$) to complete the task. The average WPM value (Equation 6.1) is **13.4 WPM** ($M = 13.4, SD = 2.4$), with a minimum observed value of **10.5 WPM** and a maximum of **18.2 WPM**. WPM, however, may be an inappropriate metric for reporting source code entry speed (also see the discussion section) but is given for completeness here.

$$WPM = \frac{|T| - 1}{S} \times 60 \times \frac{1}{5} \tag{6.1}$$

$|T|$ is the length of the transcribed string; S is the duration in seconds. A word is considered to consist of 5 characters.

A metric that is more adequate for source code could be based on *tokens* instead of words. For example, the average token length of the presented string is 3.1. Although the length is dependent on the programming language, a token may be considered shorter than the average English word. (The example code uses long identifier names and the token length is still much lower than 5). Using a token length of 3, the formula (Equation 6.2) would result in an average value of **22.4 TPM** (tokens per minute) ($M = 22.4, SD = 3.9$). A more valid value could be found by mining source code repositories and determining the average token length across main families of programming languages.

$$TPM = \frac{|T| - 1}{S} \times 60 \times \frac{1}{3} \tag{6.2}$$

Keystrokes

The metric for keystrokes per character (KSPC) measures the text entry performance as ratio of the number of needed keystrokes to the number of characters in the transcribed string (Equation 6.3). The average KSPC value of this study is **0.76** ($M = 0.76, SD = 0.09$).

$$KSPC = \frac{|IS|}{|T|} \tag{6.3}$$

$|IS|$ is the length of the input stream; $|T|$ is the length of the transcribed string.

The value reported here is the empirical, observed KSPC value and *not* the characteristic, absolute KSPC value that measures the efficiency of a text entry method based on a language model [Mac07]; the latter states the average number of keystrokes needed for each character. Here, a *keystroke* includes not only pressing a letter key but also performing a marking menu gesture, dragging the cursor, selecting a template placeholder, and inserting a suggestion from code completion. The more keystrokes needed, the higher the KSPC value. (Lower values are generally considered better.) Also, correcting errors increases KSPC since backspacing and undo actions are part of the input stream.

Error Rate

[Mac07] discusses several metrics for reporting errors. The metric reported here is the *MSD Error Rate* (Equation 6.4), which is based on the *Minimum String Distance* between the presented and transcribed string. *MSD* is also known as the *Levenshtein Distance*[3]. This value shows how similar two strings are by calculating the minimum number of editing primitives (insertions, deletions, substitutions) required to transform one string into the other [Mac07]. According to this metric, the average error rate of the study was **5%** ($M = 0.050, SD = 0.033$).

$$MSD\ Error\ Rate = \frac{MSD(P,T)}{MAX(|P|,|T|)} \tag{6.4}$$

$MSD(P,T)$ is the minimum string distance (Levenshtein distance) between the presented and the transcribed string; $MAX(|P|,|T|)$ is the larger of the length of the presented and the transcribed string.

Keyboard Configuration

All participants were introduced to keyboard properties that they could configure during the introduction phase and during the task.

The font size could be increased or decreased in 10%-steps by pressing toolbar buttons. Most participants changed the default size of 12 points to a larger value. On average, they increased the font size to 1.26 times ($M = 1.26, SD = 0.25$) the default size, which corresponds to about **15 points**.

No participant changed the **landscape orientation** of the tablet during the task. All participants left the keyboard in the **docked** state for the entire duration of the task. The keyboard was, on average, zoomed to **1.19** times ($M = 1.19, SD = 0.06$) its initial size of 1024 x 285 points. Since this average zoomed size of 1220 x 340 points (2440 x 680 pixels on the iPad's Retina display) extends the screen edges, most participants horizontally centered the keyboard so that all resized keys were visible within the screen bounds. Resizing resulted in an average key size of **13.2 mm x 12.0 mm** for the letter keys. For comparison, the default size of a letter key on the iPad's landscape keyboard is 15.2 mm x 14.2 mm; on the German keyboard, key sizes shrink to 12.9

[3] http://en.wikipedia.org/wiki/Levenshtein_distance

mm x 14.2 mm due to the extra keys for Umlauts. Figure 6.16 illustrates the average keyboard frame and key size in relation to the screen dimensions.

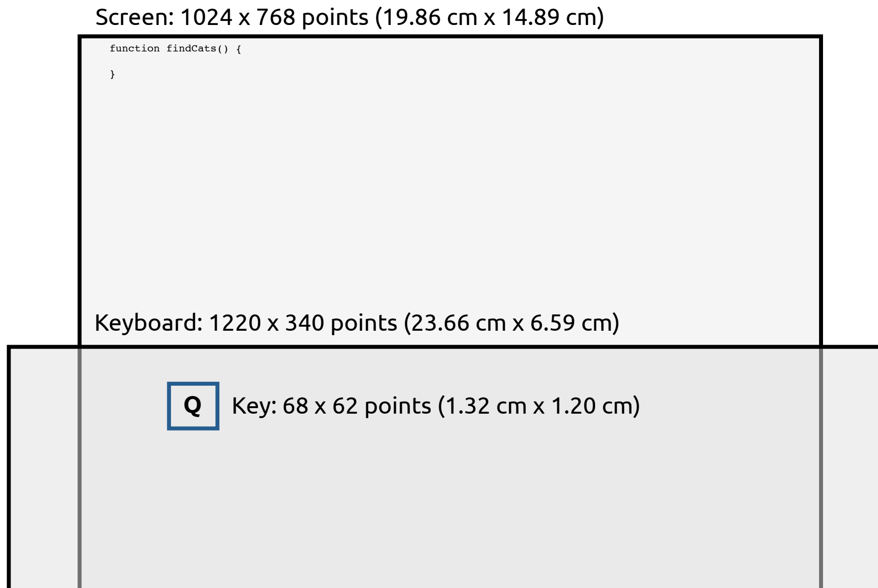


Figure 6.16: The average user-configured keyboard frame and key size in relation to the screen dimensions.

Selection Deviations

In order to determine how close participants were in pressing each key near its center point, the average target-relative deviation is calculated according to Equation 6.5, given in [Mac07]. The deviation indicates the relative distance of the touch point in relation to the center point of the key. For instance, a deviation of 0% means that each key is exactly hit at its center point; a deviation of 100% means that each key is hit at its visual bounds. In this study, the average selection deviation was **0.53 (53%)** (Figure 6.17).

$$deviation = \sum_{t \in T} \sum_{i=1}^{|P_t|} \frac{\frac{\sqrt{(x_i - x_t)^2 + (y_i - y_t)^2}}{S_t}}{\sum_{t \in T} |P_t|} \qquad (6.5)$$

t is the key (target) in the set T of all keys; P_t is the set of all touch points for t; (x_i, y_i) are the coordinates of a touch point, (x_t, y_t) are the coordinates of the center point

of the key. Since the equation abstracts a key as circle with radius S_t, non-uniform keys (*Space*, *Enter*, *ABC*) have not been considered. Letter keys have a slightly larger width than height; hence, half of the average between the width and height of a key has been used as radius.

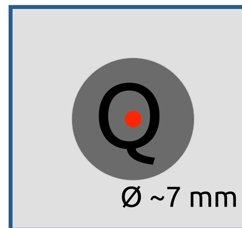


Figure 6.17: The shaded circular area shows the average key deviation in relation to the visual boundary of the key.

Figure 6.18 shows the relationship between the average selection deviation per participant and the corresponding average zoom factor of the keyboard. Since the values positively correlate ($r = 0.83, t(8) = 4.13, p < 0.01$), larger keys tend to lead to higher relative deviation and vice versa.

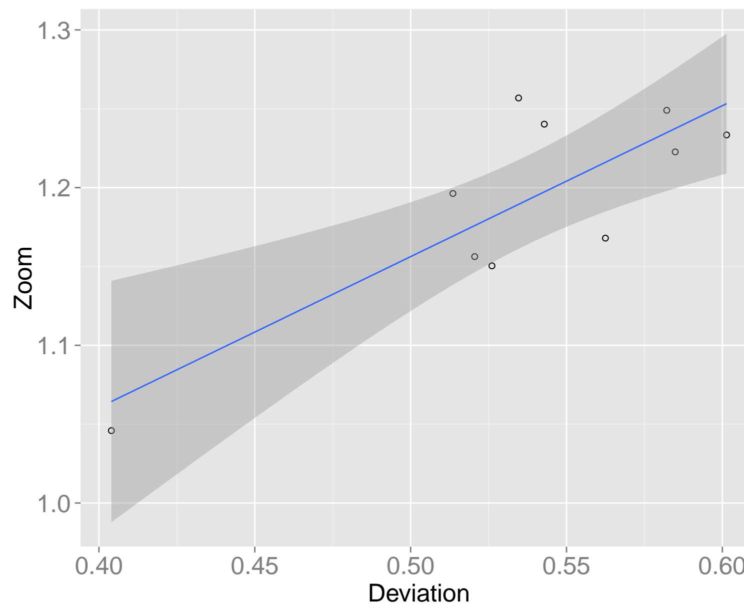


Figure 6.18: Correlation between the average selection deviation and the average zoom factor, per participant.

When the deviations are individually calculated per key, the selection points for the keys in Table 6.1 fell at least one standard deviation beyond the average deviation. The list only includes keys that were pressed at least 100 times in total. Enhancing the selection of these keys might have the largest impact on touch accuracy.

Key	Selection Deviation
Backspace	0.77
,	0.76
.	0.69
=	0.79

Table 6.1: Selection deviations for keys with total selection counts >= 100 and selection points at least one standard deviation beyond the average deviation. (Ordered by selection frequency.)

Gestures and Menu Selections

Figure 6.19 graphs the total number of selections, performed either in novice-mode or expert-mode, over time in minutes. Since the two lines do not cross in the left graph, the selection frequency increased about equally for both modes.

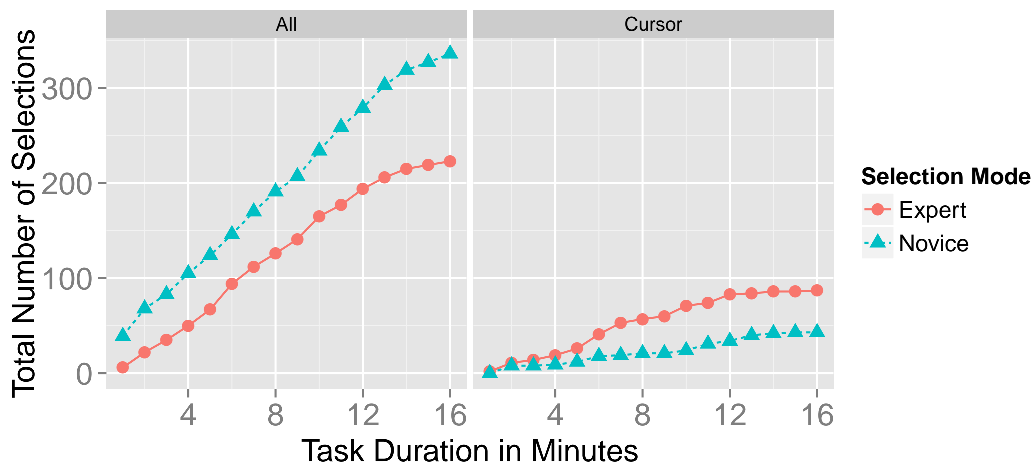


Figure 6.19: Graph of menu selections, performed either in novice-mode or expert-mode, over time in minutes. Left: for all keys; right: for the *Cursor* key. The line for expert-mode running above the line for novice-mode indicates that participants early transitioned to expert behavior with the *Cursor* key.

As for the *Cursor* key (right graph), the expert-mode line runs above the novice-mode line, indicating an early transition to expert behavior. The same effect, however, could not be observed for other keys.

Participants performed a total of **580** menu selections, consisting of **345 (60%)** selections in novice-mode and **235 (40%)** in expert-mode. Table 6.2 lists menu selections that were at least, on average, performed twice per participant (or have a relative frequency > 1%):

Menu Item	Base Key	Type	Relative Frequency (%)
;	,	Key	15.2
Function	F	Template	12.6
[]	()	Key	9.8
return	R	Template	9.5
Go Char Left	Cursor	Command	6.2
Go Line Down	Cursor	Command	5.5
for	F	Template	5.5
Go Char Right	Cursor	Command	5.0
Variable	V	Template	4.7
:	.	Key	3.8
{}	()	Key	3.8
Go Line End	Cursor	Command	3.6
if	I	Template	3.4
+	=	Key	1.9
Del. Word Before	Backspace	Command	1.9
Del. Word After	Backspace	Command	1.6
Go Line Up	Cursor	Command	1.6

Table 6.2: Relative frequencies for triggered menu items. *Base Key* is the key that triggers a *Menu Item*. *Type* is one of *Key* (a regular key), *Template* (a code template), or *Command* (a function).

As previously stated, regular keystrokes and menu gestures were disambiguated by thresholds for the distance and velocity between pressing and releasing a key. The thresholds were set to default values that seemed to work well in pre-study tests: The default value for a menu flick to be recognized as such was 50 points for distance and 100 points/s for velocity.

The average distance between pressing a regular key and lifting the finger (dkp) was **5.3** points ($M_{dkp} = 5.3, SD_{dkp} = 5.5$); the average velocity (vkp) was **67.8** points per second ($M_{vkp} = 67.8, SD_{vkp} = 78.8$). For triggering expert-mode menu selections

(i.e., without displaying the menu), the average distance (dms) was **61.3** points ($M_{dms} = 61.3, SD_{dms} = 9.5$); the average velocity (vms) was **687.6** points per second ($M_{vms} = 687.6, SD_{vms} = 313.1$).

Although these values were affected by the existing default values of the test system, the results may nevertheless be used for adjusting the defaults and evaluating them in a follow-up study. To maximize the allowed movement distance for the more important regular keystrokes, $MAX(M_{dkp}, M_{dms})$ could be set as distance threshold (~5.9 mm or about 50% of the size of an average key). To minimize the required velocity for triggering a menu selection, a reasonable upper bound of M_{vkp} could be found by adding two standard deviations: $M_{vkp} + 2SD_{vkp} = 225.4\,points/s$. All values apply to the average size of a key (see previous section).

Key Frequencies

Table 6.3 lists frequently used keys, excluding regular letter keys. Like in previous studies (see Chapter 5), *Backspace* was the most frequently pressed key. While the values for *Return* and *Shift* are perhaps expected, the relative high usage of the non-standard *Cursor* key might suggest that participants found its functionality particularly valuable.

Key	Relative Frequency (%)
Backspace	11.6
Return	4.4
Shift	2.5
Cursor	2.5
()	2.1
123	1.9
ABC	1.9
.	1.6
,	1.3
=	1.3
Undo	1.0

Table 6.3: Relative frequencies for non-letter keys with frequencies >= 1%.

Code Templates and Code Completion

The participants triggered, on average, **26.2** ($M = 26.2, SD = 8.2$) code completions and inserted **20.9** ($M = 20.9, SD = 3.1$) code templates. Table 6.4 specifies the relative frequencies of inserted code templates and their usage in relation to the actual occurrence of the template structures in the task. For instance, 100% usage indicates that participants exploited each opportunity to insert the template.

Template	Relative Frequency (%)	Usage (%)
Function Definition	34.9	100
Return Statement	26.3	91.7
For Statement	15.3	100
Variable Definition	12.9	67.5
If Statement	9.6	100
Assignment (left = right)	0.5	10.0
Function Call (functionName())	0.5	1.7

Table 6.4: Relative frequencies for triggered code templates and their usage; that is, the percentage of exploited opportunities to insert the template.

Qualitative Feedback and Observations

After the task, the participants were asked which features of the keyboard they found inconvenient and which features they found particularly convenient. Some participants also gave feedback during the task (although they were not explicitly instructed to "think aloud").

Almost all participants stated that they found code templates and code completion very convenient and time-saving. Some participants noted that they quickly became used to the menus that had been added to individual keys. Furthermore, they thought that they learned the "shortcuts" (expert-mode) during the task or that, given more time, they would quickly learn them.

Half of all participants noted that the code completion feature should insert parentheses (which the test system did not) when choosing a method, and they pointed out that method parameters should be displayed. Also, half of all participants wished that an active code template was not automatically canceled when the cursor was manually dragged to another location within the template.

Two participants considered the placement of the *Undo* key inappropriate and confused it with the position of the left *Shift* key. Another participant regarded the size of the *Tab* key as too small. Three participants mentioned their extensive use of the *Cursor* key in desktop editors and positively acknowledged its availability in the keyboard layout. One of the participants suggested to move the *Cursor* key to the lower right position of the keyboard.

Four participants sometimes experienced problems when they tried to invoke the marking menu in novice-mode. The implementation used in the study canceled novice-mode when the finger was moved while holding a key. Future implementations should thus allow some “jitter” while touching-and-holding a key. (Also see the following discussion on that issue.)

Overall, participants commented very positively on the user experience of the keyboard and remarked that it worked “surprisingly well”.

6.4.5 Discussion

The results include a number of metrics that are commonly reported in studies of text entry systems [Mac07]. However, as Kristensson et al. [KBC+13] note, “Currently there is no universally accepted experimental procedure for evaluating text entry performance”. The values given here should, therefore, be cautiously interpreted, in particular due to the different context of this study (source code vs. text). Below, I attempt to analyze notable results.

Typing Speed

Text entry speeds for on-screen keyboards tend to range from 15 to 30 WPM [Kri09]. For the iPhone, studies have stated average typing speeds of 18.5 WPM [MLC09] and 15.9 WPM [ALS10]. In another study, 15.4 WPM have been measured for a simulated iOS keyboard of a research prototype [WPR+14]. For comparison: On physical keyboards, average typing speeds of 38 to 40 WPM [Ost97] and 33 CWPM (corrected WPM) [KHHK99] have been reported; a web-based typing test reports an average value of 41 WPM[4]. However, all of these values refer to entering regular text or English phrase sets.

[4] http://www.ratatype.com/learn/average-typing-speed/

To my knowledge, no comparable values exist for entry speeds of source code on typical virtual keyboards. Despite supportive features of the CEK for entering special characters, transcription of source code is expectedly slower and possibly requires more attention than transcription of English phrases. Participants also had to continuously refer to the printed document rather than type short phrases displayed on a screen. In addition, the CEK employed no advanced touch model (and a language model only through code completion). The proposed *TPM* metric might be a more adequate measure due to the large number of short tokens in source code. Since code written in functional and scripting languages is often more compact than code in procedural and object-oriented languages [NF14], a sound *TPM* metric might need to account for differences in token length. Moreover, a follow-up study using the IDK as baseline may yield a more meaningful *relative* rather than absolute comparison of entry speed.

KSPC and Error Rate

The KSPC value being below 1 indicates that participants could take advantage of the supportive widgets for code templates, certain special characters (e.g., pairs of parentheses), and code completion. The high usage of code templates supports the usefulness of this feature. Since most of the participants' positive feedback was related to keystroke-saving features and the gesture-driven menus, the trigger mechanism through marking menus seems suitable. The KSPC value might further decrease with repeated usage of the keyboard when menu locations are internalized by users.

Although the reported error rate does not consider the exact behavior *during* error correction, the value gives an initial impression of the participants' overall accuracy. (The KSPC calculation also captures errors since keystrokes for corrections increase the value.) The measured rate of 5% appears as an acceptable value for a first version of the CEK.

Keyboard Configuration

All participants enlarged the size of the keyboard and used it in its docked state. Most participants moved the keyboard so that all keys were visible within the screen bounds. This size and position resemble, in essence, the default display area of the IDK. The default CEK area, in contrast, was apparently considered too small, suggesting that participants are used to larger keys on tablet keyboards. The average configuration

casts some doubt on the usefulness of features for resizing the keyboard and setting it into an undocked state (although one participant appreciated the possibility of docking the keyboard in his configured size). The average configured font size of 15 points points to the default size being too small. For tasks concentrating on reading source code, however, different values might apply.

Keys and Commands

As the study in Chapter 5, this study has again confirmed the importance of the *Backspace* key during code editing. Here, the *Backspace* key could delete not only the previous character but also words or lines to the left or right. On the one hand, the data shows that participants frequently used the added functionality. On the other hand, *Backspace* had the highest selection deviation; hence, the key should probably be enlarged in future revisions of the CEK. Additionally, the *Cursor* key, which is not part of standard keyboards, seemed to be popular. As one participant proposed, the key could be moved to the more expected lower right area of the keyboard. The left *Shift* key and *Undo* key should probably be swapped because two participants remarked that they would expect *Shift* as the outermost left key. Enlarging both *Shift* keys would result in a design trade-off between the number of additional keys in the same row (e.g., *Undo*) and potential accuracy gains from hitting larger keys.

Gestures and Selections

The CEK calculates the distance and velocity between touching and lifting the finger in order to disambiguate regular keystrokes and menu gestures. Finding suitable thresholds is crucial for preventing both functionalities from triggering at the same time. In order to find the "sweet spot", the proposed values should be evaluated in a follow-up study. Incidentally, the suggested distance threshold is approximately equal to half the size of a key. This value leads to a straightforward guideline for triggering the menu in expert-mode: Originating from the center of a key, the touch triggers a menu item when the finger crosses the visual boundary of the key.

Novice-mode required the finger to be held still while dwelling on the key, but the study showed that some participants performed slight "jitter" movements and thereby inadvertently canceled the menu. A more robust design should, therefore, allow slight movements while invoking novice-mode.

Both the CEK and the IDK trigger a keystroke when the finger is lifted or display pop-up menus when touching-and-holding. Contrasting physical keyboards, this design inhibits the *touched state* for resting the fingers on the keyboard. When the allowed duration of a valid keystroke is limited, the keyboard could additionally support finger resting. This idea has been closer examined by Kim et al. [KSL+13]. Their project *TapBoard* uses a short *tap* (less than 450 ms) for triggering keys and, according to the authors, participants found typing in this mode natural, and they were equally fast. To improve the interaction, a design variation of the CEK could thus utilize this threshold and the number of simultaneously pressed keys.

UX and Summary

The overall design approach of adding frequently needed characters to the layout, reducing layout switches, adding supportive widgets, and supporting fast command triggers through marking menus has been well received by users. They could work with the familiar *QWERTY* layout as basis and quickly learned the added functionality after only few minutes of training time. Improvements to this design should primarily be directed towards supporting a more advanced touch model and language model. Furthermore, while *smart typing* was partially supported by menu items, *code hints* (see the introductory section) is another useful code creation method that could be added to the keyboard.

6.5 Improvements and Simulations

In this section, I propose specific enhancements to the CEK, based on the presented study results. Enhancements include a revision of the key layout, touch model, and language model. Also, first simulations of the new models demonstrate their potential benefits.

6.5.1 Key Layout

Figure 6.20 shows a revised key layout of the CEK. Compared to the previous version, the following keys have been changed:

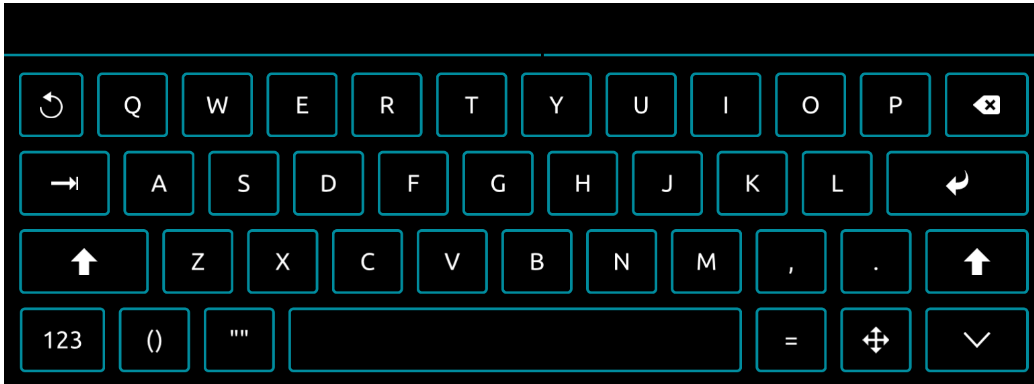


Figure 6.20: A revised key layout for the CEK.

- The *Search* key, inactive during the user study, has been removed. Displaying the search widget does not necessarily require a dedicated trigger key. Since the widget would be displayed in the top bar of the keyboard, a button or icon placed near this area might be more suitable.
- The *Cursor* key has been moved to the lower right area of the keyboard and thus closer aligns with the *Arrow* keys on physical keyboards. A trade-off of this position is the clipped display of the lower menu items in novice-mode. In order to reduce the issue, the bottommost point of the menu could be shifted to the bottom screen edge so that all items become visible. In addition, the menu displays a tooltip at the top of the screen when the users moves the finger towards a menu slice. (This feature was already active during the user study.)
- The *Undo* key has been placed at the top left position. Some participants confused *Undo* and *Shift* during the study. Moreover, *Undo* (or *Back*) tends to be placed at the top and left areas in desktop applications.
- *Backspace*, both *Shift* keys, *Tab*, *Special*, and *Dock/Undock* have been enlarged. The increased area might better account for larger selection deviations at the leftmost and rightmost positions.

Since one of the strengths of virtual keyboards is their customizability, certain choices (e.g., the placement and mapping of special keys or menu items) are left up to the user by providing a configuration file or GUI. Users might quickly learn key locations and exploit directional flick gestures, regardless of their initial configuration.

6.5.2 Touch Model

A more advanced touch model for the CEK comprises two main parts: Improving the accuracy for hitting keys and improving the interaction (i.e., the disambiguation between keystrokes and menu gestures).

Accuracy

Previously, the CEK merely used the visual boundary to determine the target key for a touch point. A slightly better approach could determine the target key based on the closest distance between the touch point on the keyboard area and the center point of a key. More advanced approaches may either use machine learning techniques or utilize additional sensor data about touches. However, machine learning techniques require training data; extended information (e.g., posture, size, or angle) about finger touches is often not available on commodity tablets. An approach that is solely based on positional data about touches and target sizes has recently been suggested in [BZ13]: The *Bayesian Touch Criterion* (BTC) is a statistically derived criterion for finger touches and has been shown to significantly improve selection accuracy. For completeness, Equation 6.6 shows the calculation:

$$\begin{aligned} BTD_2(s,t) = \frac{1}{2}\left[\frac{(s_x-c_x)^2}{0.0075d^2+1.68}+\frac{(s_y-c_y)^2}{0.0108d^2+1.33}\right]+ \\ \frac{1}{2}ln(0.0075d^2+1.68)+\frac{1}{2}ln(0.0108d^2+1.33) \end{aligned} \quad (6.6)$$

The equation already contains experimentally measured constant values, see [BZ13] for details of the derivation. (s_x, s_y) is the touch point s; (c_x, c_y) is the center point of the target t; d is the diameter of t. All units are millimeters. The result of the calculation is the *Bayesian Touch Distance* (BTD). The target for a touch is then found by calculating BTD for each target and choosing the target with the lowest value.

Figure 6.21 shows 100000 randomly placed touch points distributed over a number of CEK keys. The keys are abstracted as circles, with their diameter set to the width of a key plus half of the gap between keys. BTC is used for calculating the closest (randomly colored) target key for each touch point.

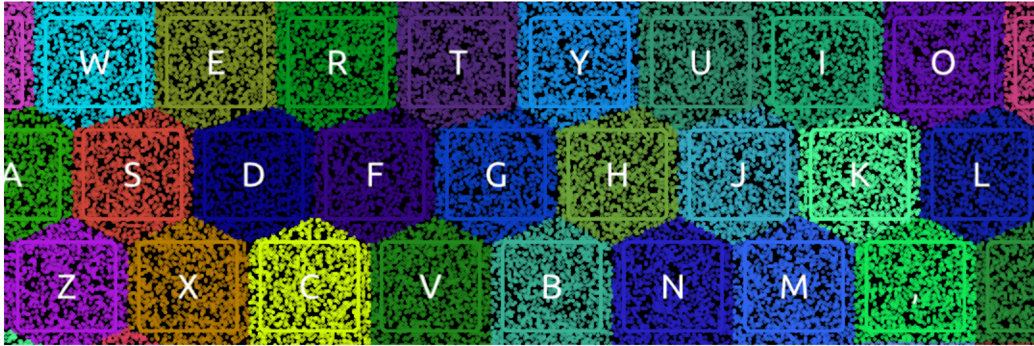


Figure 6.21: Simulation of the *Bayesian Touch Criterion* applied to the CEK. The graphic shows 100000 randomly placed touch points and their target keys.

Applying BTC results in gapless hit areas according to the BTD calculation. In its current form, this approach could be used for enhancing the hit areas of keys that can be abstracted as circles. Since non-uniform keys have larger hit areas, not using BTC for these keys and falling back to a closest-centroid approach might be acceptable. The exact differences in hit accuracy between showing and hiding a rectangular border around keys should be separately measured.

Interaction

As pointed out in the discussion, the disambiguation between keystrokes and gestures could be improved while also allowing for finger resting. A revised version could implement the interaction according to the following rules (also see Figure 6.22):

- When only one finger is touching-and-holding and the key provides a menu, display the menu and cancel the keystroke. Also allow small "jitter" movements while holding the key. If the key does not provide a menu, do nothing.
- When multiple fingers are simultaneously touching-and-holding, cancel all actions and do nothing.
- When any finger is lifted, trigger the keystroke only if the time between pressing and lifting the finger does not exceed a threshold of 450 ms [KSL+13].
- When the visual boundary of the key is crossed with a velocity of 225 points/s, trigger an existing menu item in expert-mode. (This threshold should be refined after further studies.)

A limitation of this design is the lacking support for repeating a key while touching-and-holding. The CEK version of the study, however, did not support key repeats, and no participant pointed it out as missing feature. (The IDK also does not support key repeats.)

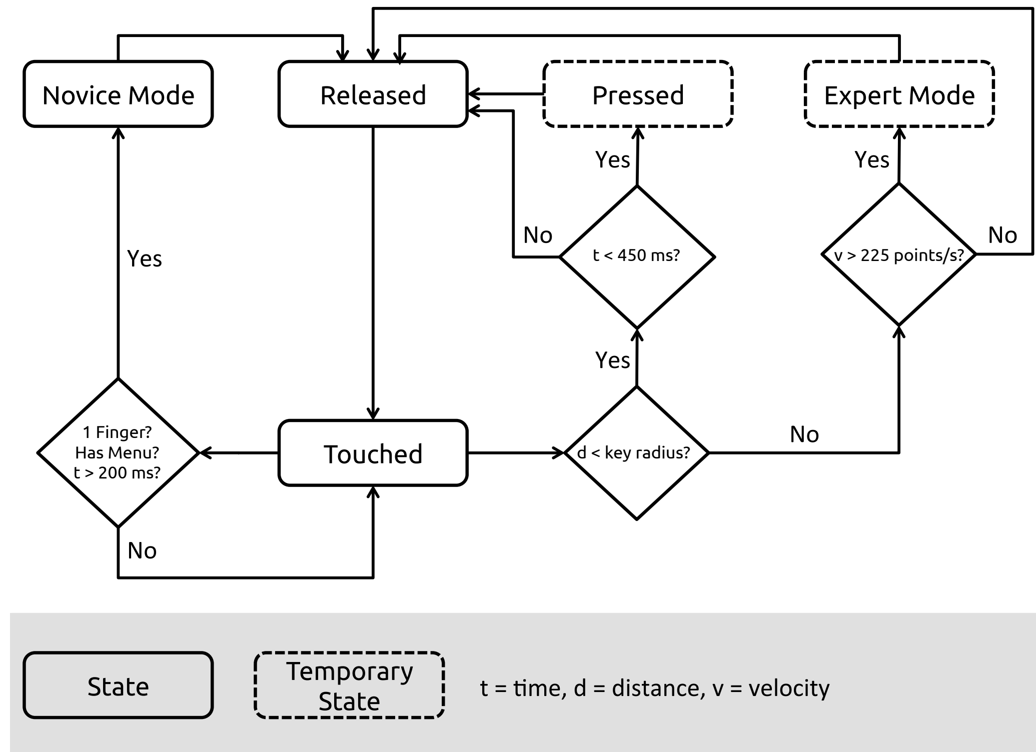


Figure 6.22: A refined key interaction model for the CEK.

6.5.3 Language Model

Modeling language for text entry systems usually involves statistical procedures applied to a *vocabulary* or *lexicon* to predict which words the user might enter next. As the user is typing, the model is queried for likely matches, which are presented as selectable suggestions in the user interface. The ability to predict what the user might type next can be obtained from *n-gram* models: *Markov chains* store occurrences for each word and its following word. Simple predictions then consider the previous word in order to calculate the likelihood for the next word. More advanced *bi-gram* models that take the likelihood of word pairs into account are common for supporting European

languages [Mac07]. Furthermore, static offline models are often combined with *online adaptive techniques*, based on the assumption that users tend to reuse entered words [Mac07]. According to Tanaka-Ishii [Mac07], an ideal language model combines both offline and online techniques:

- "A large general corpus for developing the static initial language model. [...]"
- "A document of specific context to adapt the initial language model dynamically."

The Language Model of Source Code

As far as *source code* is concerned, the language model typically consists of a syntax tree and the user's editing context. IDE modules utilize the data to provide editor services such as code completion or refactoring. Since code completion engines both consider the programming language and continually analyze entered code, it could be argued that they realize both of the above-mentioned components of a robust language model. The degree of "intelligence" built into these engines, however, varies substantially between IDEs. Without smart code completion, using SDKs and libraries with their ever-growing number of methods and properties can impede developer productivity.

Most open-source or commercial engines analyze types, previously defined identifiers, variable scope, and other heuristics for filtering the list of possible matches. This reduced list is presented to the programmer, who can either select the topmost suggestion, navigate to a different entry, or refine the list with additional keystrokes. The most likely items should be promoted to the top of the list to reduce the decision and selection time. Often, however, the list is only alphabetically sorted. Eclipse, for instance, takes a hybrid approach: Entries are alphabetically sorted, but when sufficient context information is available (e.g., the expected return type), individual entries are sorted to the top [HP11]. Researchers have proposed more elaborate solutions to predict likely completions, for example by mining code examples for popular items [MFSM10], identifying emergent programming practices of crowds [FSW+14], or considering only the lines of code directly before the editing point [ARSH14].

Language-agnostic Code Completion

Due to the lack of static type information, smart code completion for *dynamic* languages, such as JavaScript or Python, has been much more challenging. Nevertheless, the goal here is to improve code completion *without* resorting to static type information and in *language-agnostic* ways. Supporting code intelligence for multiple programming languages requires considerable implementation effort; this cost can be reduced by employing language-agnostic approaches.

Basic code completion capabilities could be provided–without consideration of any semantics–by simple indexing of all identifiers and keywords in a source file. To increase the quality of suggestions, completion engines for dynamic languages can utilize *type inference*; that is, attempting to infer the runtime type of a value (e.g., a variable value) at any given point in time. Inferring types, however, again introduces *language-specific* code intelligence into the completion system.

Due to these issues, the approach suggested here treats an existing code completion engine as "black box": The engine is only expected to return a list of sorted proposals, which could either be generated by basic identifier indexing or sophisticated type inference. The client of the engine adds the following two improvements as *complementary* layers:

1. **Predictions**: The most likely next suggestion is based on the user's personal code completion history. Predictions are sorted to the top of the list and thus facilitate quicker selection and saving of keystrokes. The rest of the list is sorted according to the internal process of the completion engine.
2. **Spelling Correction**: Despite a more fault-tolerant touch model, users may still make mistakes when entering code. Surprisingly, most existing code completion engines fail when misspelled code is entered. Adding a spelling correction layer returns basic proposals even when the completion engine fails to return any results.

Predictions

Hindle et al. [HBS$^+$12] have found that "a high degree of local repetitiveness, or regularity, is present in code corpora and, furthermore, that n-gram models effectively capture these local regularities". Programmers frequently reuse identifiers, similarly

to how writers reuse phrases when writing a text. Applied to code completions, this insight suggests that a programmer's locally completed code could serve as basis for generating predictions of future completions. Research has shown that code completion systems that use historical data from a user's editing session can outperform default algorithms, even when no additional type information is available [RL08].

Predictions could be generated by *Markov chains*, or by only taking the most frequently used or most recently used items into account. An algorithm that combines multiple factors has been proposed in [FC12]: *AccessRank* not only uses *Markov chains*, but also the combined frequency and recency of previously visited items. Additionally, *AccessRank* incorporates a configurable threshold that either favors the *stability* or *accuracy* of the generated prediction list. On the one hand, maximizing stability is particularly desirable in UIs where frequent reordering of list items should be minimized; the positions of already learned item locations should change as little as possible so as to not interfere with usability. On the other hand, highly ranked predictions should be sorted as far as possible to the top of the list to enable quick selection. Dependent on the application domain, this threshold can be configured accordingly.

Equation 6.7 shows the main components of *AccessRank*. (For details on how the individual components are calculated, see [FC12].)

$$w_n = {w_{m_n}}^{\alpha} {w_{crf_n}}^{\frac{1}{\alpha}} w_{t_n} \tag{6.7}$$

w_n is the *AccessRank score* of an item; w_{m_n} is the Markov weight; w_{crf_n} is the weight for the combined frequency and recency (CRF); w_{t_n} is the time weight, which considers the time and day of item access. α is an empirically determined threshold for weighting the Markov and CRF components. After calculating the *AccessRank score* for each item, an additional pass compares item pairs and determines if one item is allowed to overtake another item in rank based on a *switching threshold* (also empirically determined).

The time weighting component is optional and can be turned off to achieve performance improvements. Code completions, for instance, are probably less sensitive to the exact time and day of access than other item types. Although implementing *AcessRank* is more time-consuming than implementing comparable algorithms, its predictions

have been shown to be more accurate and stable than those of other approaches [FC12].

Using *AccessRank* for code completions involves recording an *item visit* when the user selects a completion from the list. As a consequence, *user-initiated* completions become the basis for predictions rather than already existing tokens in the code.

Spelling Correction

A spelling correction layer could either correct only common programming language keywords (e.g., “while” or “self”), or all locally entered keywords and identifiers. A combination of mostly language-agnostic tokens and local tokens might yield the best results. For instance, when the user starts typing “finc”, the spelling corrector should suggest “func” or “function”; when the user types “slef”, it should suggest “self”; when “car” is entered to refer to an existing local token “cat”, the spelling corrector should suggest the token.

The spelling correction layer may only need to be queried when the code completion engine fails to return any results; otherwise the interaction between code completion and spelling correction could introduce ambiguity. (Although Bi et al. [BOZ14] have demonstrated that both completion and correction accuracy can be maximized, it is questionable if the added complexity could be justified in a programming environment.)

Standard algorithms should be sufficient to generate useful corrections for the mentioned use cases. Most spell checkers rank suggestions by calculating the *Damerau–Levensthein distance*[5] from the entered word to words in the lexicon. The approaches mostly differ in their use of efficient data structures for achieving the best runtime performance with a large lexicon.

Combining Predictions and Spelling Correction

Figure 6.23 illustrates the revised language model for code completion. The previously explained layers for predictions and spelling correction are intended to operate on top of an existing code completion engine.

[5] `http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance`

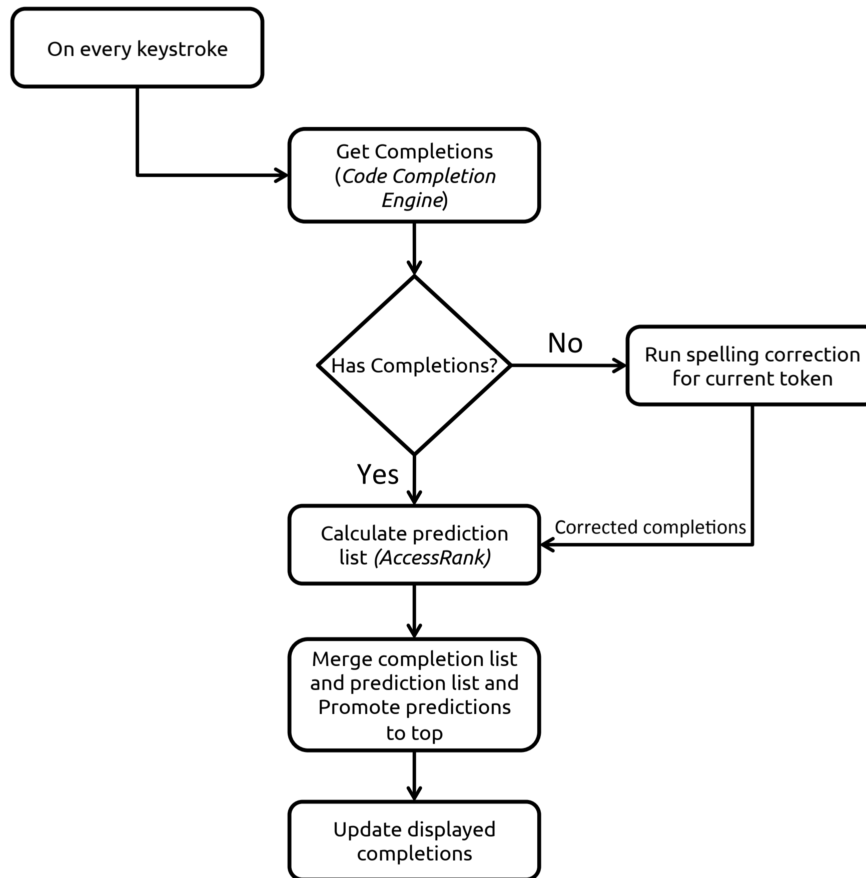


Figure 6.23: The proposed model for code completions in the CEK.

Simulation

To measure the effectiveness of this revised code completion model, I have simulated both the old and the new approaches using the source code of the task given in the user study. The simulation is based on automatically triggering code completions for a "perfectly entered" program (i.e., each character is linearly replayed as it occurs in the source code; see the discussion below on limitations of this method). To keep the simulation simple, the spelling correction layer has not been added.

Both methods have been compared according to the following procedure:

1. Find all tokens for potential code completions in the source code (i.e., all keywords and identifiers).

2. Linearly replay each character of the program and trigger code completion when the first character of a token is encountered.

3. Check if the current token is found in the list of completions and record its rank.

4. Calculate the mean and standard deviation of the rank totals.

The mean rank values and standard deviations for the two versions are shown in Table 6.5. The average rank for the old approach is **2.54** and **1.61** for the new approach. Lower ranks (and therefore higher positions in the displayed list) are achieved when the predictions are merged into the list of completions. With an extended version of the source code containing 80 instead of 50 lines of code, the difference between the old and new approaches becomes even larger (**2.74** and **1.45**, respectively). The simulation used the standard *AccessRank* configuration for medium list stability and prediction accuracy.

Type	LOC	Completed Tokens	Rank (M)	Rank (SD)
Regular	50	85	2.54	3.12
with Predictions	50	85	**1.61**	3.23
Regular	80	168	2.74	3.42
with Predictions	80	168	**1.45**	3.11

Table 6.5: Mean ranks and standard deviations for code completions with and without predictions.

By replaying the source code and automatically invoking the code completion engine at the positions of tokens, the effects of different code completion strategies could be isolated and compared. However, this approach has several limitations: First, it does not accommodate for programmers non-linearly editing incomplete code or triggering completions at other character positions of a token. Second, it neglects usability issues such as searching and selecting an entry in the displayed list. Third, the code sample is small and might not be representative. A thorough study would need to involve humans for more realistic measures.

The goal of this initial simulation was to validate the proposed model at a basic level. The results seem to suggest that local AccessRank-based predictions for reused tokens improve the average rank of suggestions. Moreover, this enhancement can be implemented in a language-agnostic way on top of existing code completion engines, the predictions are fast to compute, and no complex analysis- or change-tracking-infrastructure is required.

6.5.4 Widgets

In previous sections, I have presented improvements to the key layout, touch model, and language model of the CEK. In this section, I suggest enhancements to the UI widgets displayed at the top area of the keyboard.

Code Completion

Programmers frequently employ the code completion mechanism as quick way for browsing documentation. Most editors display abbreviated documentation in a pop-up next to the selected proposal. In the CEK, the completion list is "always on" and proposals are inserted when the user taps on an item. Documentation could be integrated by pressing an icon next to each item or by performing a gesture. For instance, a swipe-up gesture over an item could invoke a small pop-up above the item. Tapping another item while the pop-up is visible then updates the documentation until the pop-up is dismissed.

Code Templates

The CEK version of the user study displayed items for navigating template placeholders. When no template was active, this space remained unused. The empty area could be used for *predicting* templates (again using *AccessRank*). For example, when the user has repeatedly created an *if template* within a *for template*, *AccessRank* captures this transition and next time suggests an *if template* after an inserted *for template*. (Although the current editing context could additionally be considered, this would most likely introduce language-specific functionality.)

6.5.5 A Revised Model

Extending the basic model shown in the introduction of this chapter, the refined model illustrated in Figure 6.24 summarizes all previously discussed components. It comprises three main layers that aim at representing a hardware- and language-agnostic approach to code entry on touchscreens.

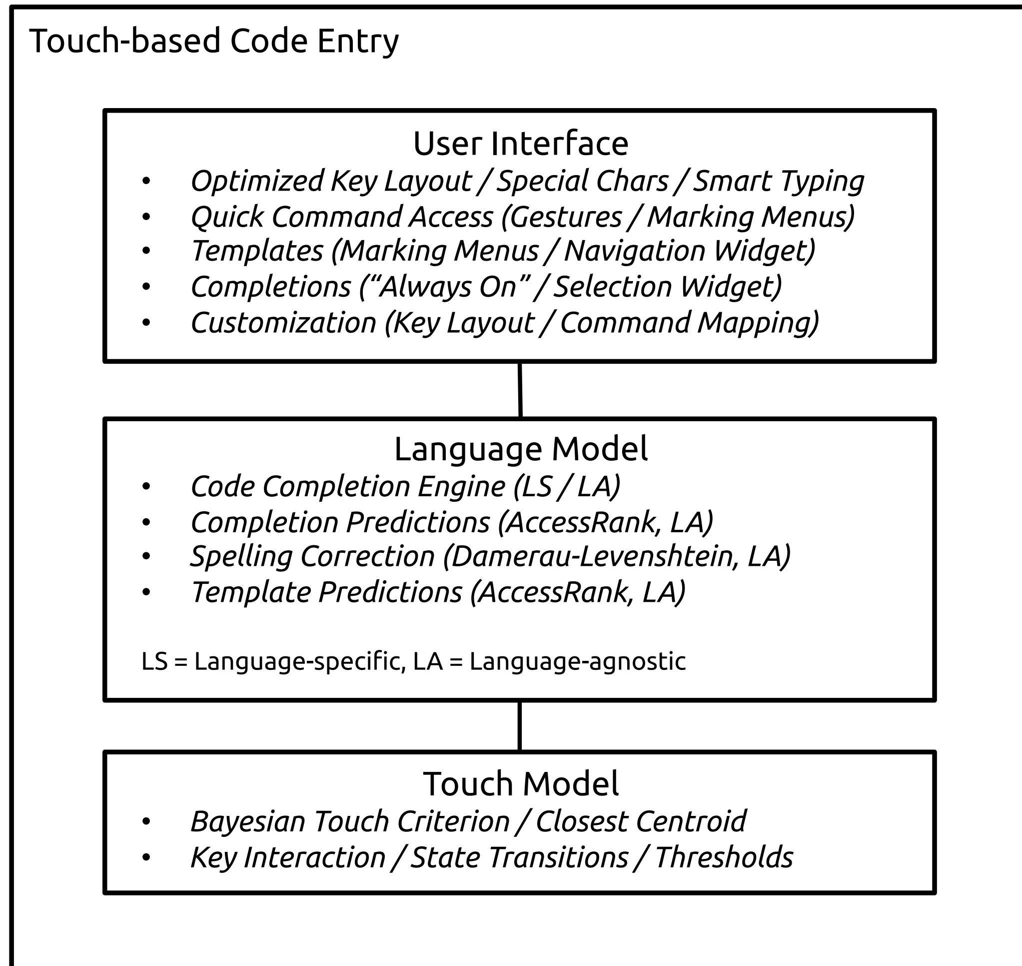


Figure 6.24: A revised model for code entry on touchscreens.

6.6 Conclusion

In previous chapters, I have addressed interaction techniques for editing and selecting source code. In this chapter, I have focused on methods for *creating* source code on touchscreens. The first section has introduced the challenges involved with entering text on a touchscreen using a virtual keyboard. Text entry has long been an important HCI research area. Previous efforts, however, have largely been directed towards optimizing the input of English phrases. Source code differs in that special characters should be quick to access, whole code structures should be easy to create, and code intelligence functionality should support the programmer.

The second section has listed five common ways of how these goals are achieved in desktop editors. Having detailed their mechanics, I have presented the design of a custom keyboard for code entry (CEK). This keyboard provides an optimized layout, enables fast gesture-driven access to special characters and commands, and renders supportive UI widgets for controlling two essential productivity features, namely code completion and template insertion.

Furthermore, the CEK was subject of user study on code entry. I have reported several text entry metrics and other interaction-related measures. The positive user feedback and the fact that the first version only implemented a rudimentary touch model and language model have motivated further research on improvements. Following the user study, I have suggested specific enhancements concerning the key layout, touch model, language model, and UI widgets. Simulations of the new models have given first insights into their effects on a future version of the CEK.

Finally, I have shown a revised model for code entry on touchscreens. Future research efforts could primarily address three areas: First, the enhancements should be evaluated in a larger follow-up study, ideally involving experts. Second, other code creation mechanism besides code templates and code completions should be examined. Third, in order to better assess text entry speeds, the CEK should be compared to a baseline keyboard such as the IDK.

Although virtual keyboards still lack the haptics and preciseness of physical keyboards, I have attempted to demonstrate how appropriate interaction techniques and algorithms mitigate the limitations of existing systems.

Part III

Design and Implementation

Chapter 7

A Touch-enabled IDE

This chapter presents the design of a coherent system integrating the previously studied interaction techniques for touch-centric code editing and their supporting IDE modules. The first part discusses platform choices and the devised approach to interaction design for gestures, commands, and menus. The second part introduces concrete modules for file browsing, working set management, code navigation, code entry and editing, and code review. While the following chapter discusses technical aspects and software architecture, this chapter focuses on the interactive behavior and elaborates on the rationale behind all design decisions.

7.1 Device Class and Platform

The proliferation of devices with differently sized touchscreens and the variety of operating systems have led to numerous possible configurations for designing and implementing touch-enabled software. Consequently, the choice of a particular *device class* and *OS* frequently precedes concrete design work. Recent attempts at enabling *adaptive* layouts that dynamically adjust to changed screen properties have reduced the technical implementation effort, but human judgment is required to instruct the system *how* to adapt the UI to the new environment. Furthermore, as applications grow in complexity and increase their use of custom-developed UI elements, standard frameworks tend to fall short. This is particularly true for cross-platform development tools, which often trade user experience (e.g., platform conventions or performance) for reduced deployment effort. While popular platforms, such as Android, iOS, or

Windows Mobile, may be equally suitable for achieving successful user experience, the type of device might be the more decisive factor as far as the present work is concerned.

Touchscreens are available in a variety of sizes, ranging from tiny smartwatch screens to smartphone screens, tablets, tabletops, and large display walls. Common sense might suggest that extreme screen sizes (i.e., smartwatches or large display walls) are inappropriate for typical programming tasks. The issue becomes less clear, however, when considering the medium-sized range including smartphones, tablets, and tabletops. The project *TouchDevelop* [TMdHF11] (see Chapter 3) has demonstrated that programming on a smartphone screen is technically feasible by introducing a special-purpose programming language. The interaction techniques proposed in this work permit more flexible editing in a mainstream programming language but might be degraded or rendered unusable on small smartphone screens. Tabletops, on the other hand, provide sufficient screen space but may be most appealing in collaborative work scenarios. Also, time-consuming editing tasks, performed in a top-down view of source code on the surface, could be strenuous. (Chapter 9 revisits potential uses of tabletops for software development tasks.)

In a study about the impact of screen sizes on usability, Raptis et al. [RTKS13] have found that "users who interact with larger than 4.3in screens are more efficient during information seeking tasks". Although their own study investigated devices only up to a screen size of 5.3 inches, the authors refer to a prior study incorporating mobile screens up to 9.7 inches and state, "the largest screen led to higher participants' enjoyment, while the smaller screen-size elicited greater perceived mobility". 4.3 inches could be regarded as typical size for a modern smartphone, while 9.7 inches, for instance, is the size of the iPad tablet. The user interface and interaction surface required for programming might impose stricter limits regarding small screen sizes. Hence, the design and interaction methods presented here are intended to work best on tablet-sized screens (∼7 inches and above).

7.2 Approach to Interaction Design

The following sections describe the general approach of this work for enabling gesture-driven interaction and invoking commands.

7.2.1 Integration of Gestures

Designing a touch-based system involves consideration of the extent to which gesture-driven interaction should be supported. Conservative approaches primarily support tap-based interaction where users point on the visible elements of the user interface (e.g., buttons or list items). Although a *tap* could be classified as discrete gesture, modern UIs frequently exploit continuous movements, such as *swipe* or *pinch* gestures. A *swipe*, for instance, is often used for navigating list entries back and forth since the required movement appears to map naturally to the behavior of the UI widget. The advantages of this *direct manipulation* are perhaps most evident in the ubiquitous *pan* gesture for scrolling content. Less familiar gestures include motions consisting of multiple strokes or shapes. For example, users could draw an “X” on the screen to close the current view of the application. On the one hand, such designs may contribute to an expressive gesture vocabulary; on the other hand, as the number of actions increases, finding matching motions and shapes can become difficult and may result in arbitrary or ambiguous mappings between actions and their triggers.

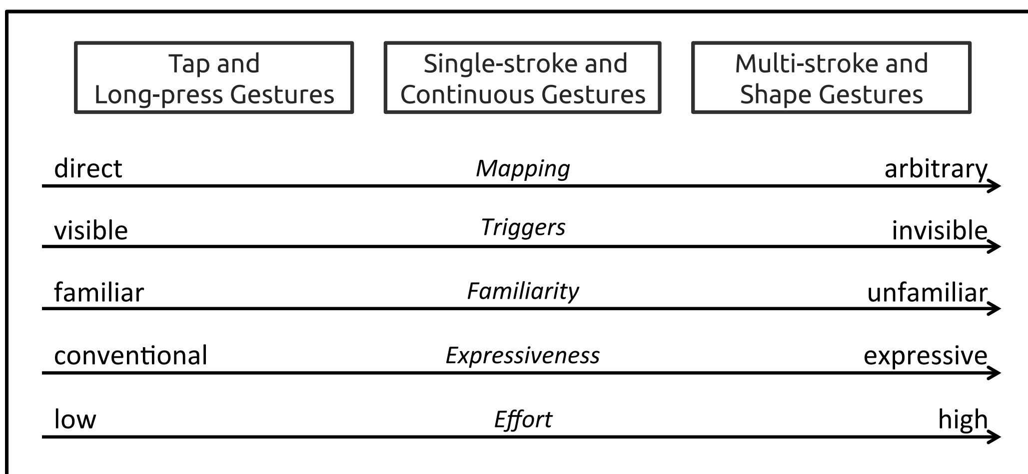


Figure 7.1: Properties for varying levels of gesture-driven interaction.

Figure 7.1 illustrates properties for varying degrees of gesture-driven control. It is important to note that these properties are not to be understood absolute but rather as tendency. Depending on the context and task, for instance, a *swipe* may be *more* expressive than a multi-stroke gesture. In the study presented in Chapter 4, users designed gestures from each of the three categories, but they were cautious employing movements from the category of “Multi-stroke and Shape Gestures“. Due to the issues

of finding adequate mappings for large command sets, this category of gestures has not been considered in the present work.

7.2.2 Conflict Resolution

Attaching gestural interaction to UI elements can quickly lead to conflicts. As previously stated (Chapter 4), user-elicited gestures are particularly susceptible to this issue since subtle ambiguities only emerge in fully working applications. User-defined gestures then compete with the platform's built-in gestures or the interaction of specific UI widgets. This is why standard mobile frameworks provide developers with fine control over event processing and gesture recognition, which enable them to react to and resolve such conflicts. The following example (Figure 7.2) demonstrates a concrete use case requiring methods for resolution:

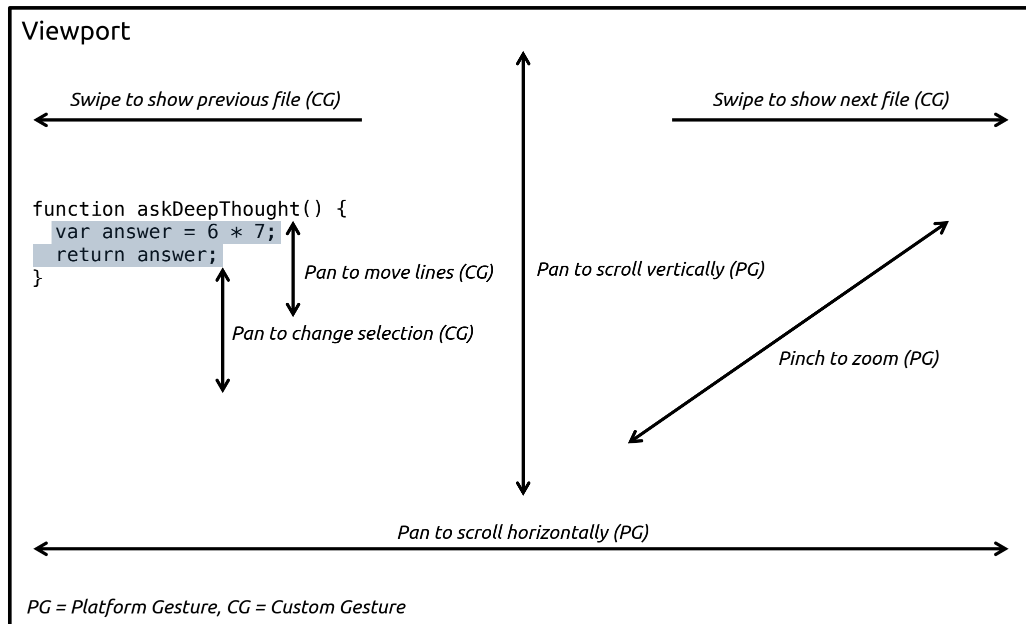


Figure 7.2: Examples of competing gestures in the editor viewport.

The editor renders a scrollable viewport that displays the source code of a file. According to the conventions of most mobile platforms, scrolling is performed through horizontal and vertical *pan* gestures. If the designer of a gestural system intends to add support for file navigation through horizontal *swipe* gestures, he might overlook that simultaneously supporting a *pan* and *swipe* gesture results in a conflict: The *swipe*

gesture cannot be performed without triggering a *pan* gesture. In this scenario, the conflict causes the unintended side effect of horizontally scrolling the source code. This example is only one of several situations where gestures require additional *disambiguation*. Other potential conflicts are generated by selecting code, changing the selection range, or by applying operations such as moving selected lines to a different location (Figure 7.2).

As a consequence, this work has mainly employed the following strategies for handling competing gestures and ambiguities:

Target Region
: A gesture that is performed in a designated area of the screen overrides a gesture attached to the underlying larger area. For example, performing a *pan* gesture on the right edge of the editor viewport triggers *interpolation* and *outline* scrolling (see section 7.3).

Exclusive Gestures
: Certain gestures may block other gestures from being simultaneously processed. For example, when users perform a *pan* gesture to modify the range of a syntax-aware selection gesture (see Chapter 5), the standard gesture for viewport scrolling is disabled during the operation.

Number of Touches
: The number of simultaneous touch points distinguishes custom gestures from standard gestures that should not be overridden. For example, performing a *pan* gesture using two fingers moves selected lines, whereas performing the same gesture using one finger triggers the default behavior (viewport scrolling).

Thresholds
: Gestures are initiated when configured thresholds for the distance and velocity of the movement are reached. For example, when the user performs a *flick* gesture over a keyboard key, the default keypress is canceled and instead a code template will be inserted (see Chapter 6).

All other gestures are recognized as follows:

Target Element
: Gestures are attached to elements of the user interface. The gesture is recognized when it is initiated within the bounding box of the element.

Configurable Properties

Configurable properties of a gesture determine the action to perform; that is, *pan* gestures are distinguished by their direction (horizontal vs. vertical movements), *tap* gestures by the number of taps, and *touch-and-hold* gestures by their dwelling duration.

Editor State

The state of the editor document (text buffer) or viewport determines the type of gesture. For example, a *touch-and-hold* gesture over selected source code triggers a contextual menu, whereas the same gesture over non-selected code initiates a selection operation.

7.2.3 Widget-based Techniques and Menus

Although an application could support a considerable amount of gestures by means of disambiguation methods, another known issue associated with gesture-driven interfaces is their tendency to lack appropriate *disclosure* mechanisms. Users first need to discover the presence of a gestural trigger for an action, but without the application providing explicit hints, important features might remain unused. Hence, interfaces should either integrate clues for the user or employ *widgets* for specific actions. For instance, some of the selection techniques presented in Chapter 5 render visual elements instead of exclusively relying on gestures.

Marking menus, a well-researched method combining both gestures and widgets, have been discussed earlier (Chapter 3). The custom keyboard shown in Chapter 6 has implemented this technique for accelerated access to the commands of individual keys. Since users gradually transition from novices to experts when they repeatedly invoke a command, they only need to be instructed how to initiate the menu itself. In addition, this work has integrated three enhancements that optimize *marking menus* for code editing operations and global actions:

Repeatable Actions

Some code editing operations (e.g., indenting code) tend to be performed in quick succession, but repeatedly executing these operations can be onerous. Items of commands that have been marked as "repeatable" fade out after execution (Figure 7.3) and allow for repeated tapping on the icon during the animation. Each tap resets the fade-out animation, thus enabling any number

of repetitions. The animation quickly finishes so that visual distractions are minimized.

Chorded Taps

As alternative to the space-consuming display of hierarchical marking menus, additional menu items can be shown by performing a tap with the second finger. As the second finger touches the surface, the first eight items are replaced with new items. Each tap cycles through a new menu level, replacing the previous one. This design also allows for quick chorded selection: When the menu locations have been internalized, the first finger can be released as soon as the second finger touches the surface and thus triggers the item in the respective menu slice. This interaction shows some similarity to *FastTap* [GCS+14] (also see Chapter 3). On small screens, cycling through the *breadth* of the menu might be more convenient than navigating the *depth* of multi-level marking menus.

Global Feedback

A global *tooltip* at the top of the screen displays the command names for a short duration. This feedback has been added for three reasons: First, to reveal the command names of items that are occluded by the user's hands; second, to disclose the labels for icon-based elements; third, to confirm expert-mode selections. In novice-mode, the tooltip is updated as the user hovers over an item of the menu.

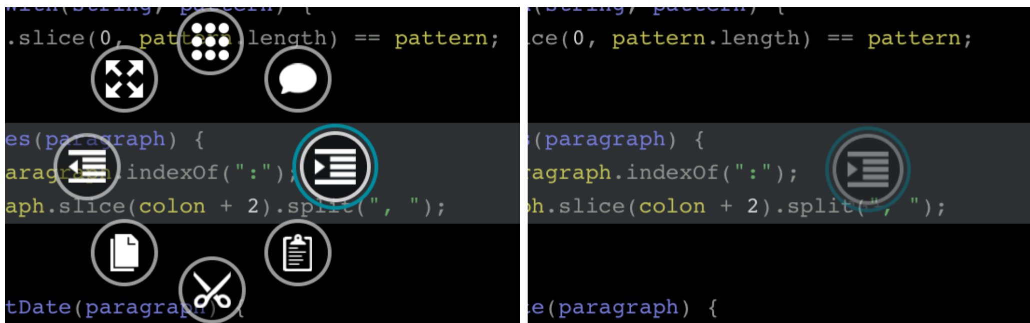


Figure 7.3: Marking menus optimized for code editing operations. Left: Marking menu in novice-mode with the item for *code indentation* highlighted. Hovering the item triggers feedback in a global tooltip at the top of the screen. Performing taps with the second finger cycles through additional menu levels. Right: Actions can be repeated by tapping on an item during the fade-out animation. Each tap resets the animation and re-executes the command.

7.2.4 General Guidelines

Overall, the interaction design of the IDE adheres to the following guidelines:

1. Favor familiar gestures: Most actions are performed through tap-based gestures and conventional continuous gestures (see Figure 7.1); multi-stroke and shape-based gestures have not been exploited in this work.
2. Minimize overriding of platform gestures: Custom gestures do not override or overload standard gestures provided by the platform. Conflict resolution, if required, is realized through the aforementioned techniques.
3. Add special-purpose widgets: Widgets reduce the disclosure problem and the use of arbitrary or ambiguous gestures. Examples of this work include selection widgets (see Chapter 5) or marking menus, tailored to code editing operations (see previous section).

7.3 IDE Components

The sections below present the design and its rationale for main IDE modules, including file browsing, working set management, code navigation, code entry and editing, and code review.

7.3.1 File Browsing

Most IDEs provide file browsing facilities, typically shown in a separate panel on the right or left side of the screen. The hierarchical structure of folders and files is represented as *tree view*; folders can be expanded or collapsed, and files are opened by mouse-click or keyboard shortcut. On small touchscreens, expanding multiple levels of the tree consumes considerable screen space and requires horizontal scrolling to display clipped content. Hence, mobile applications frequently display only a single level of the hierarchy, replacing the previous level when opening a folder and displaying a button for navigation to the previous level.

Here, the file browser is displayed in a *drawer* (i.e., a sliding panel) that is revealed when the gutter of the code editor is horizontally dragged with a *pan* gesture. The same interaction in the reverse direction closes the drawer and maximizes the available

area of the code editor. In the drawer area, navigation is performed through *tap* and *swipe* gestures: A *tap* onto a folder navigates one level deeper, whereas horizontal *swipe* gestures navigate to the previous or next level–similarly to the history concept of a web browser. Navigating back from any level to the root level is accelerated by a *two-finger-swipe* gesture. While a *tap* gesture on a file displays its content in the first editor pane, a *touch-and-hold* gesture initiates a *drag-and-drop* operation that lets users assign the file to any editor pane (see the next section).

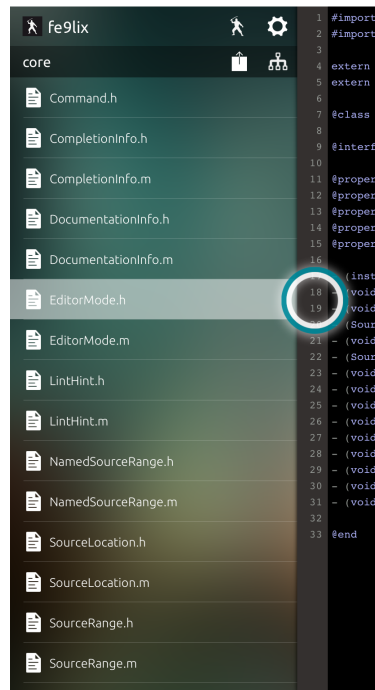


Figure 7.4: File browsing *drawer* with support for the following gestures: *horizontal pan*, *one-finger-swipe*, *two-finger-swipe*, *tap*, and *touch-and-hold*. Here, the user has started dragging a file into the adjacent editor pane. Feedback in form of a colored, pulsating circle indicates valid targets for the drag-and-drop operation.

7.3.2 Working Sets, File Sets, and Layouts

Solving a programming problem or working on a maintenance task often requires repeated switching between multiple files [KMCA06]. IDEs have included various features to assist developers in organizing files into cohesive sets. For example, *tabs* let users quickly change to opened files. Editor windows can be further subdivided into *split views* for side-by-side presentation of code. The Eclipse IDE allows users to group

files into *working sets* that filter folders and files and thus hide unnecessary artifacts. More advanced organizational forms can be added through plugins that attempt to restore a developer's context based on metrics gathered from code editing sessions (see Chapter 3).

This work implements the concept of *working sets*, *file sets*, and editor *layouts*: A working set contains one or more file sets. File sets are collections of individual files, bound to the view of an editor layout. Working sets can be be quickly switched through a gesture-driven widget. File sets replace tabs and add *predictions* for files that are likely to be selected next. Layouts are created by naturally mapping gestures to zoomable *split view* arrangements.

The widget for managing working sets is inspired by the *Patchworks* code editor [HF14], which has been shown to increase navigation speed and reduce navigational errors. The key idea of *Patchworks* is a linear stripe ("patch grid") that is constrained to left and right movements for fast switching to related code fragments (see Chapter 3). Here, the technique is modified to be compatible with editor layouts (instead of code fragments) and enhanced with gestural interaction.

Working Sets

When the user performs a *pinch-out* gesture over the editor area (reinforcing the metaphor of zooming out to an overview representation), a semi-transparent horizontal list of working sets is shown (Figure 7.5). The list can be infinitely scrolled to the left and right with *pan* gestures (continuous scrolling) or *swipe* gestures (discrete scrolling). Unused slots are initially empty. The active working set is visually highlighted and centered when the widget is shown. Tapping on an item changes to the working set and, if the slot is empty, implicitly creates it at the selected position. Alternatively, a *pinch-in* gesture hides the widget without changing the active working set. For easier identification, users can add labels by *double-tapping* into the field below a slot, followed by entering a short name. A working set is deleted by performing a *swipe* gesture from right to left over the label area and confirming by *tapping* onto the shown icon. After deletion, the slot is displayed as empty and can be reused. Since the list provides an infinite number of slots, users are free to assign working sets to any contiguous or non-contiguous sections of the list.

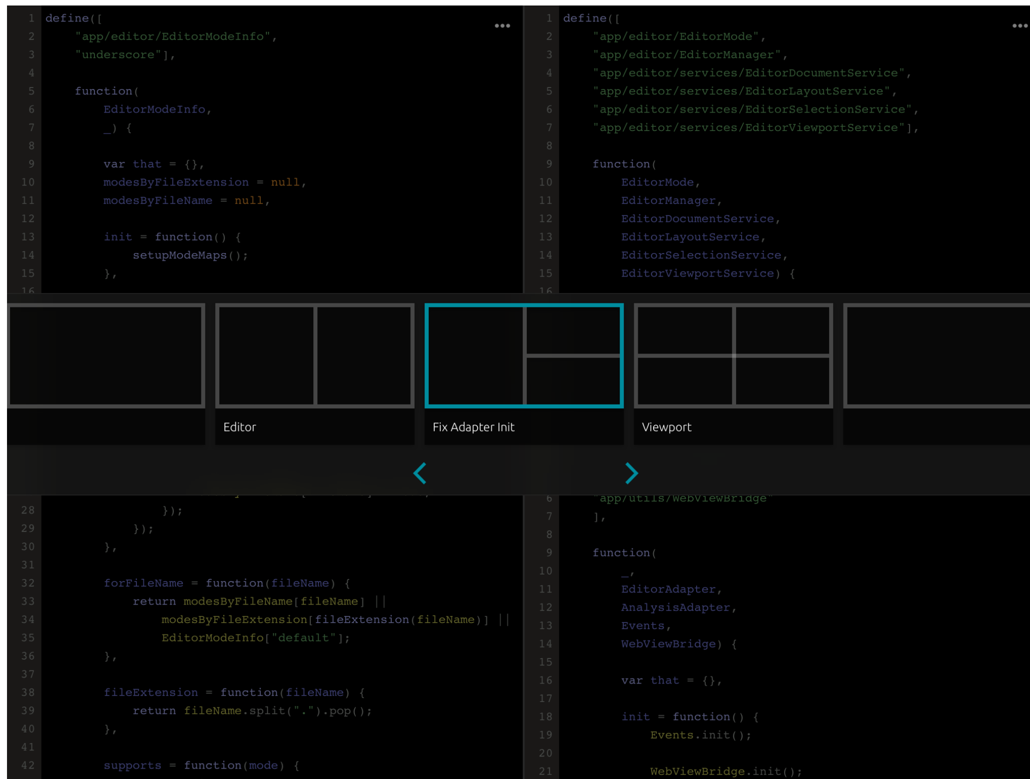


Figure 7.5: Widget for switching working sets. Performing a *pinch-out* gesture reveals a semi-transparent overlay for changing working sets through *pan*, *swipe*, and *tap* gestures. Working sets can be organized in slots along an infinitely scrolling horizontal list.

Editor Layouts

An editor layout is a particular split view arrangement for side-by-side views of code. A layout is assigned to a working set by drawing simple directional strokes that resemble the desired arrangement. For instance, the layout is vertically split by performing a *vertical swipe* gesture over the slot of the working set. A layout with a vertical split and an additional horizontal split in the right half is created by two *swipes*: a *vertical swipe*, followed by a *horizontal swipe* in the right half. The split view arrangement is visualized through respective lines displayed over the working set (Figure 7.6). As the user switches through layouts, the configuration is directly reflected in the editor panes below the semi-transparent overlay. File sets associated with an editor pane (see below) maintain their position in changed layouts. If the new layout dismisses certain panes, their files move to the last visible pane. The layout widget supports

eight different configurations, requiring users to perform at most two *swipe* gestures that naturally map to the arrangement. After selecting a layout, each pane of the split view arrangement can be zoomed through *pinch* gestures, which allow users to quickly maximize the editing area or move to an adjacent editor pane.

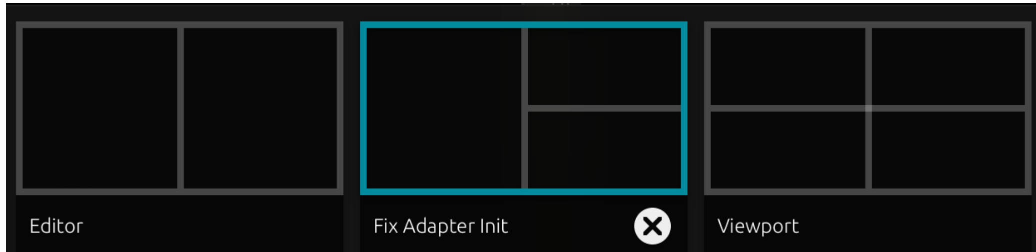


Figure 7.6: Gesture-driven editor layouts. Each working set can be split into horizontal and vertical editor panes. Layouts are switched by performing, at most, two simple directional *swipe* gestures over a working set. The highlighted split view arrangement, for instance, is created by performing a *vertical swipe*, followed by a *horizontal swipe* in the right half of the item. Layouts are reset by performing an invalid gesture or by *swiping* right to left and *tapping* on the displayed delete-icon.

File Sets

File sets are implicitly created when users drop files from the drawer into an editor pane. Instead of rendering a conventional tab bar, the collection of files is displayed in a *pop-over* view (Figure 7.7), which is opened by *tapping* on an icon at the top of the editor pane. A pop-over is a temporary view that floats above other views and disappears as soon as the user taps outside of its bounding box. The view embeds a control bar and a vertically scrolling list. In contrast to horizontal tabs, a vertical list supports browsing through a larger number of files. The control bar at the top allows users to organize files (e.g., removing individual files or emptying the list). In order to accelerate file switching, the list is split into two halves: The bottom half lists files based on the *recency* of access (i.e., the most recently accessed file is displayed at the top); the top half uses the *AccessRank* algorithm [FC12] (see Chapter 6) to predict files that the user might select next.

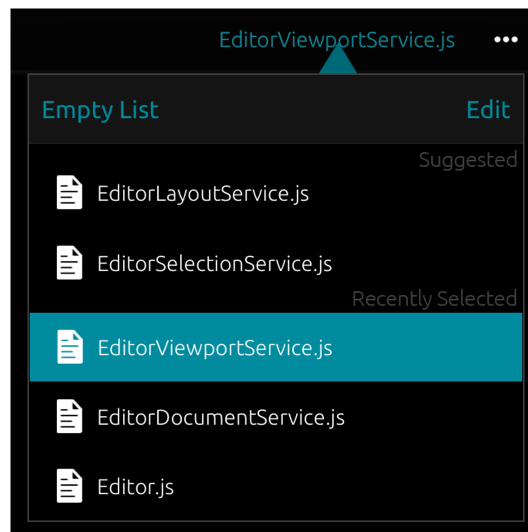


Figure 7.7: *Pop-over* views containing file sets. File sets replace tabs and embed vertically scrolling lists for browsing collections of files associated with an editor pane. To accelerate file navigation, the list is split into *recency-based* entries (bottom) and *AccessRank*-based predictions (top). Labels shown on the right mark the two sections. A control bar at the top of the list shows options for removing individual entries or emptying the list.

Discussion

Working sets, *file sets*, and *layouts* allow developers to collect all relevant code for a task and reduce the navigational effort of switching between sets of related artifacts. The importance for such organizational tools has previously been emphasized and confirmed by other studies [KAM05a, KMCA06]. In contrast to prior work, this work has not implemented the concept of *code fragments* (i.e., small units of code). Code is organized at the granularity of files, but, as shown in the following section, supported by tools for navigating to specific locations *within* files. In the initial version of the working sets module, I experimented with a design where working sets could be freely arranged on a 2D canvas. However, personal tests have confirmed the finding of recent research [HF14]: Instead of burdening users with manual positioning and sizing of individual views, providing tools that are intentionally *constrained* can lead to higher efficiency. Here, constraints are established by a *Patchworks*-inspired widget for organizing working sets along a single horizontal dimension, and by predefined layouts that can be switched through natural gestures.

In view of the lack of screen space and keyboard shortcuts, it is essential to offer an alternate mechanism for file switching. In desktop IDEs, file switching is often realized via the file browser, tabs, or dialogs for quickly opening files by name. Mobile platforms, in contrast, have provided different widgets such as pop-overs. To accelerate file navigation in these views, I have added recency-based lists and AccessRank-based predictions. (Predictions could, as an extension, also be applied to working sets.)

The shown functionality has not included options for file *creation*. A perhaps obvious choice is creating files via an "add" or "+" button placed in the toolbar at the top of the editor pane. To support gestures, interaction techniques incorporating the *bezel* of the device could be employed [HYP+10]: Users could cross the bezel and then continue dragging a file icon into one of the visible editor panes. In touch-based applications, this interaction may reinforce the process of "making something out of nothing" [HYP+10].

7.3.3 Navigation

As developers navigate code, they form a mental model of the program; this mental model, however, does not necessarily match the hierarchical organization of files or code structures (e.g., inheritance) [SES05]. Consequently, hierarchical file browsing facilities are insufficient for supporting developers in navigating along their "mental path". In this section, I present both the *inter-file* and *intra-file* navigational tools of the IDE.

Inter-file Navigation

The features for working sets, file sets, and layouts are mainly organizational instruments promoting *inter-file* navigation. Working sets switch collections of file sets, and file sets ease revisiting files. *Recency-based* lists, displayed in the lower part of the file set widget (Figure 7.7), can provide for up to 70% of navigations between files [PG06]. In addition, they allow users to browse *all* previously accessed documents. *Predictions*, displayed in the upper part, incorporate recency as factor but also consider the transitions between documents, as well as temporal data [FC12].

Access to a file set is accelerated through gestures. While the widget could be accessed by tapping on the name of the current file displayed in the toolbar, this interaction

requires up to three taps for file navigation: (1) showing the editor toolbar, if required, by tapping on the icon in the upper right corner of the editor pane; (2) tapping on the name of the current file displayed in the toolbar; (3) tapping on an entry in the file set. The following alternative methods are available to reduce the number of required taps (Figure 7.8):

- *Temporary Trigger*: A *touch-and-hold* gesture on the toolbar-toggle temporarily shows the file set. The widget is shown only as long as the finger is touching the surface. As the user continues *panning* downwards, the file list highlights the corresponding entries. An entry is selected after releasing the finger. Instead of three separate *taps*, this interaction requires only a combined *touch-and-hold* and *pan* gesture. Also, it allows users to access the file set when the editor toolbar is hidden.
- *Local Gestures*: The toolbar-toggle serves as visual "anchor" for initiating *flick* gestures. A *left-flick* gesture starting at the icon navigates to the previous file of the history and a *right-flick* to the next file. The same gestures can be performed over the current name of the file when the toolbar is visible.

Figure 7.8: Gesture-driven file navigation. Local gestures are assigned to the visual elements of the editor toolbar: *Swipe* gestures over the file name or toggle-icon navigate through the history. A *touch-and-hold* gesture on the toggle-icon, followed by a *pan* gesture, selects a file from the temporarily displayed file set widget (see Figure 7.7). (Arrows and circle only added for emphasizing target elements.)

Code editors frequently support jumping between edit locations, thus mixing *inter-file* and *intra-file* navigation. The VIM editor, for example, maintains different lists for recording locations: The *jump list* records motions that move between and within files; in contrast, the *change list* records locations of edits that were applied to the current text buffer. Here, global gestures, performed in the editor viewport, let users navigate the file set history. In addition, each file restores its previous scroll position and cursor location (see section 7.3.4). Since one-finger and two-finger gestures are already assigned to other functionalities (panning the viewport and *Undo/Redo*, respectively), file set navigation is disambiguated by horizontal *three-finger-swipes*.

Intra-file Navigation

Desktop editors are equipped with various methods for *intra-file* navigation, including viewport scrolling, jumping to specific line numbers and method names, or moving to related code fragments (e.g., the definition of a focused method). In this work, users scroll the contents of the text buffer through the familiar *pan* gesture. However, scrolling long files by repeatedly *panning* the viewport is onerous. In order to accelerate moving to relative sections of the file, such as the upper, middle, or lower part, the editor viewport reserves an invisible area for *interpolation scrolling*: When the user *pans* over the right border of the editor–commonly occupied by a scroll bar in desktop editors–the touch position is relatively mapped to the scroll position of the viewport. This interaction allows users to move rapidly through the file or jump to a particular section. Tapping into the invisible area causes brief visual highlighting as feedback and disclosure mechanism.

Interpolation scrolling is extended by a gesture-controlled widget for navigating at the method-level. Research has found that methods assume a key role for efficient source code navigation. For example, method-level navigation gave users advantages when they attempted to reach the target of a navigation task [HF14]. Furthermore, users tend to focus method signatures more than other source code elements [RMM+14]. Navigation to methods is often accomplished by utilizing the concept of a document *outline*, generated by a code analysis module that identifies all methods in the file. IDEs such as Eclipse render outlines in a separate panel that is permanently displayed or temporarily invoked through a keyboard shortcut. Here, the outline is displayed by using the same interactive zone as for *interpolation scrolling*: Instead of directly starting the *pan* gesture, the user first performs a *touch-and-hold* gesture and then continues interpolated *panning* over the list. As the finger touches the name of a method, the editor viewport is scrolled to the method and highlights its signature (Figure 7.9). Since the widget is shown during scrolling and hidden when the finger is released, this interaction requires only *one* continuous motion to locate a method quickly.

Other *intra-file* navigation methods, for example moving to related code fragments, are realized through menu-based techniques (see the following sections). Menus reduce the need to introduce arbitrary gestures for navigation actions that seem hard to map to directional movements (previous/next or up/down).

Figure 7.9: Interpolation scrolling and outline for *intra-file* navigation. The interaction of locating a specific method consists of *one* continuous motion: (1) performing a *touch-and-hold* gesture over the right edge of an editor pane reveals a temporary outline listing all methods in the file; (2) interpolated *panning* lets users rapidly move to a certain method; (3) releasing the finger hides the widget. Scrolling to a relative section of the code is performed by directly *panning* over the right editor edge (i.e., without displaying the outline). The touch location is relatively mapped to the scroll position of the viewport as the user pans within the height of the editor pane.

7.3.4 Code Entry and Editing

The design of code entry and editing functionality builds upon the study results of previous chapters. This section shows the implementation, a new cursor concept, and the interaction between viewport scrolling and layout zooming.

The Cursor

Unlike the default behavior of text views in mobile systems, tapping into the editor area does *not* automatically invoke the keyboard; instead, a *tap* gesture only sets the location of a cursor representation. The rationale behind this design lies in the common assumption that code tends to be more read then edited. This reasoning is also apparent in editors such as VIM that, by default, sets its mode to “normal mode” rather than “insert mode”. While the latter is only explicitly invoked for entering and

modifying characters, the former is the starting condition for operations and their associated motions. On space-constrained touchscreens, hiding the keyboard until needed reduces unnecessary obstruction of the editor content. Furthermore, users can execute a number of editing commands without having to resort to the keyboard (e.g., moving, deleting, or formatting code).

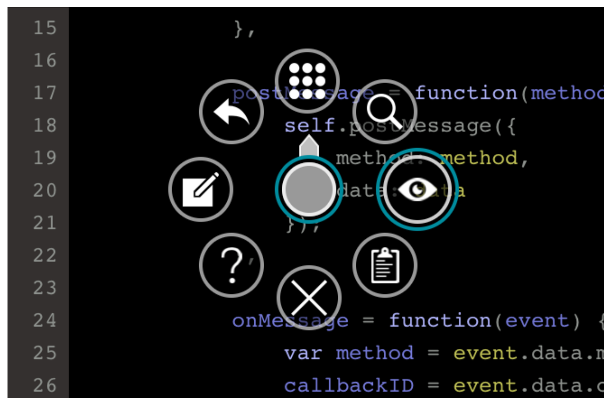


Figure 7.10: Cursor representation and context menu. The cursor shows a sufficiently large semi-transparent handle for dragging. *Double-tapping* toggles the keyboard visibility and a *touch-and-hold* gesture opens a marking menu. The menu shows items for actions that can be applied to the current cursor position or the token under the cursor, here (clockwise, starting at the item in the north-direction): Toggling the keyboard, searching for the focused token, highlighting references, pasting from the clipboard, removing the cursor, showing type information and documentation, renaming, and jumping to the definition of the focused token. The menu interaction works according to the description in section 7.2.3.

The cursor can be dragged using its handle and snaps to the nearest character position when it is released. *Double-tapping* the circular area toggles the visibility of the keyboard. A *touch-and-hold* gesture opens a context menu (i.e., novice-mode of a marking menu, Figure 7.10). The cursor does not integrate expert-mode of marking menus (see section 7.2.3) due to a conflict between their *flick* gestures and the *pan* gesture for cursor dragging. However, the delay for novice-mode is set to a tolerable minimum of about 200 ms.

Code Selection and Context Menus

The interaction techniques for code selection have been detailed in Chapter 5. Figure 7.11 shows a selected line, displaying magnet handles and adjustment handles at the

anchor and head positions. Gestures and widgets allow for *syntax-aware* selection; *selection rails* ease the selection of block statements and facilitate extending the range to parent nodes.

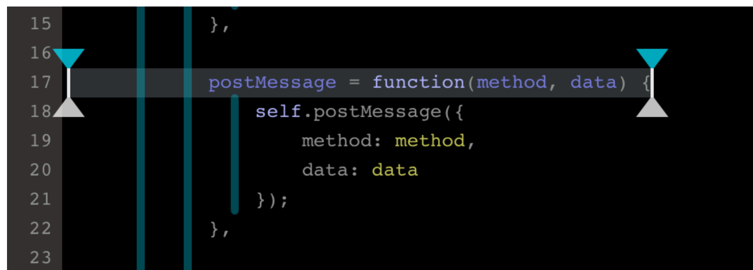


Figure 7.11: Selection handles and *selection rails*. Selection handles enable both *syntax-aware selection* (top handle) and fine adjustments (bottom handle). *Selection rails* facilitate selecting block statements and their parent nodes. See Chapter 5 for details on the interaction design.

A *touch-and-hold* gesture over a selected range of code opens a context menu (Figure 7.12).

Figure 7.12: Context menu for selections. The menu shows items for actions that can be applied to the current selection, here (clockwise, starting at the item in the north-direction): Toggling the keyboard, toggling comments, indenting, pasting from the clipboard, cutting, copying, outdenting, and extending the selection. The menu interaction works according to the description in section 3.2.3.

Similar to the cursor menu, the context menu for an active selection cannot be triggered without causing a conflict: A *flick* gesture over the selected range would simultaneously cause scrolling of the viewport. Therefore, the menu enables alternative invocation through a *two-finger-tap* gesture, which circumvents the delay of novice-mode and thus accelerates the execution of a menu command. This gesture is comparable to

the right-click on desktop systems or the identical gesture on Mac OS for triggering contextual actions.

Duplicating and moving lines can either be achieved through contextual cut/copy-paste actions or through faster gesture-driven methods as described in Chapter 4. However, integrating the suggested interaction would again lead to a conflict with viewport scrolling, which–according to the guidelines stated earlier–takes precedence over other actions. Hence, the actions of duplicating and moving lines are distinguished by the number of touch points: *Two-finger-panning* duplicates lines as proposed; holding down a third finger at the target location sets a temporary mode for cutting instead of copying the lines. (Desktop systems differentiate such actions through a modifier key.) Since this interaction is not obvious, an appropriate disclosure mechanism should be added. Alternatively, an icon shown at the target location could let the user choose between copying and cutting.

Code Entry, Scrolling, and Zooming

Users can enter code via the code entry keyboard (CEK) presented in Chapter 6. When the CEK is docked at the bottom of the screen, the editor area is shifted upwards, and the cursor is updated in the focused area as the user types. The CEK integrates typical Emacs-type [Fin91] movement commands so that users do not have to reach to the cursor for repositioning (see Chapter 6).

Two buttons placed in the editor toolbar let users adjust the font size per editor pane. Panes with small font size configurations can be temporarily enlarged to ease interacting with the source code: The editor layout as a whole serves as *canvas* that can be quickly maximized through a *pinch* gesture. The combination of font size adjustment at the editor level and fluid zooming at the layout level enables flexible arrangements for side-by-side code interaction and navigation. The *one-finger-pan* gesture changes between viewport scrolling and layout panning, depending on the scroll position of the viewport. This behavior ensures that all code is viewable when the editor layout is zoomed. Figure 7.13 illustrates a zoomed editor layout with differently configured editor panes.

The frequently used *Undo/Redo* commands are available via the CEK and via global gestures, which may be used when the keyboard is hidden. Left and right *two-finger-*

swipes performed over the editor viewport disambiguate these actions from viewport scrolling and layout panning.

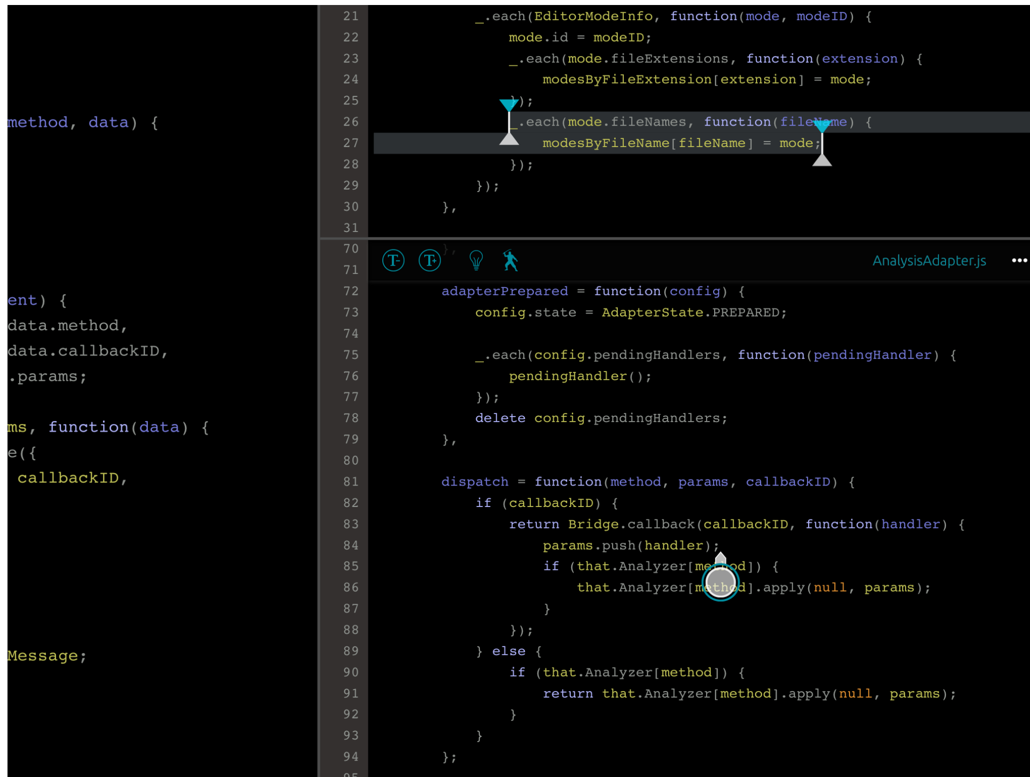


Figure 7.13: Zoomed editor layout with differently configured editor panes. The layout consists of a vertical split and a horizontal split in the right half. The left pane is configured with the default font size, whereas the top right and bottom right panes have been configured with smaller font sizes. To temporarily increase the interaction surface of the bottom right pane, the user has zoomed and panned the editor layout as a whole through a single *pinch* gesture. A *one-finger-pan* gesture scrolls the viewport within the editor pane. When the viewport has reached its minimum or maximum scroll position, the same gesture moves the editor layout to reveal clipped content.

7.3.5 Error Highlighting and Code Review

This section shows how errors or warnings from static analysis are displayed to the user. Following that, I describe how the system enables marking problematic code via the built-in code review facility.

Error Highlighting

Desktop IDEs have employed a combination of techniques to present programming errors to the user. Typically, an error is shown in the editor gutter as icon, underlined in the source code, and reported in a separate IDE panel. In addition to syntactic and semantic errors, IDEs frequently show warnings for violations of code conventions or other hints generated by external plugins. For interpreted programming languages, *linting* (i.e., the identification of problematic code through static analysis), has been a popular tool to detect syntactic discrepancies or bad practices according to configured rules.

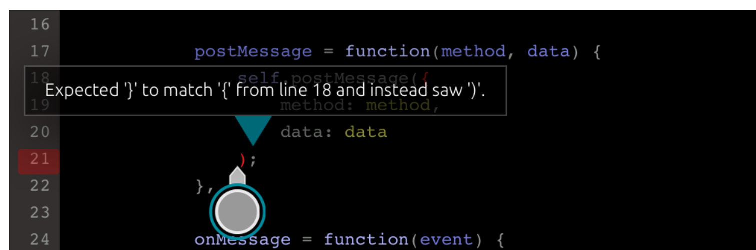


Figure 7.14: Error highlighting. Errors and warnings are shown as colored semi-transparent overlays in the gutter area. Repeated *tapping* on the overlay cycles through all errors per line. Individual errors are displayed as pop-over view that contains a description and points at the exact source location.

This work integrates the results of a *linter* into the editor gutter and adds interactivity for highlighting errors or warnings at the exact source location. Via a toggle-button in the editor toolbar, the user can opt-in to display errors in the gutter. A tooltip, briefly displayed at the top of the screen, indicates the number of errors and warnings in the file. Errors and warnings are shown as colored semi-transparent overlays (red and yellow, respectively) over the line numbers. A *tap* gesture on the overlay displays a description of the error in a pop-over view pointing directly at the source location (Figure 7.14). If a single line contains multiple errors, repeated tapping cycles through all errors. A tap outside of the pop-over area dismisses the view.

Code Review

Contrasting programs that automatically detect errors in the background, review tools let users mark problematic source code *manually*. The process of *code review* has been accepted as effective way to reduce software defects. Developers frequently resort to

external applications for conducting reviews because such functionality has not been part of the common IDE feature set. Mobile touch-based platforms, on the other hand, might be well-suited to support this task, which largely consists of reading, navigating, and marking source code.

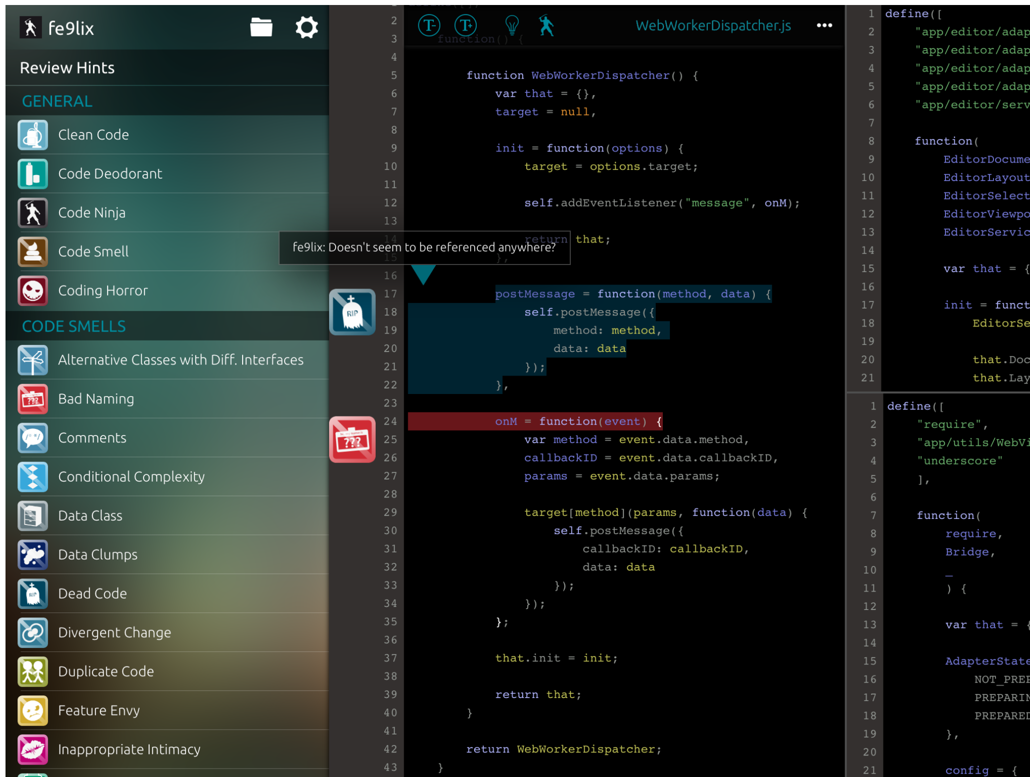


Figure 7.15: Code review facility. *Review hints*, represented by icons, can be dragged from a global list onto selected code and then appear in the extended area of the editor gutter. A *tap* gesture on an icon toggles code highlighting and user comments; a *swipe-left* gesture lets users remove the hint after confirmation; a *swipe-right* or *touch-and-hold* gesture reveals all stacked hints as horizontal list.

Here, code review is integrated alongside linting. A toggle-button in the editor toolbar lets users horizontally extend the gutter area. This extension offers additional space for displaying and interacting with *review hints*. A built-in collection of hints for common flaws in source code is accessible by revealing the file browser (see section 7.3.1) and switching to the list via a toolbar button. Each *review hint* is represented by an icon and an associated color. When the list is open, users can *touch-and-hold* on an item to initiate a drag operation and then drop the hint over selected code. The marked code is temporarily highlighted in the respective color, and the icon is displayed in the

extended gutter area. A *tap* gesture on the gutter-icon toggles the code highlighting and a tooltip containing user comments (Figure 7.15). A *swipe-left* gesture over the icon removes the hint after confirming with a tap on the displayed delete-icon. If more than one hint is assigned to a particular line, the hints form a stack. The items of the stack can be revealed as horizontal list by performing a *swipe-right* or a *touch-and-hold* gesture on the icon.

The collection is divided into "general-purpose" hints and specific *code smells*. The latter has been introduced by Fowler [Fow99] in the context of *refactoring* (see Chapter 4). Code smells indicate bad practices that can be resolved through appropriate refactorings. The set of icons was originally developed for two other projects arising from this work [Raa12a, RFW12] (see Chapter 9). Each code smell is uniquely identified by an icon, thereby helping developers to recognize potential flaws quickly. The functionality described here could be further extended by integrating a *collaborative workflow* (see Chapter 9). Moreover, the hints could be linked to the commenting or issue tracking features of modern version control platforms.

7.4 Conclusion

In this chapter, I have introduced the interaction design for modules of a coherent touch-based IDE. The system integrates methods that have emerged from the user studies of this work, as well as various supporting modules. The latter include file browsing, working set management, code navigation, code entry and editing, and code review. It is important to note that–within the scope of the present work–a number of essential IDE facilities (e.g., deployment or debugging) could not be considered. It remains open to examining how such components could be optimized for touch interaction.

In realizing the specified facilities, I have attempted to base design decisions on a number of general guidelines. The multitude of available commands in IDEs, coupled with the absence of keyboard shortcuts, calls for interaction methods that ease command invocation while taking advantage of touch interaction. Exclusively relying on gestures can lead to issues associated with finding appropriate mappings and resolving ambiguities. Hence, I have mainly employed a combination of familiar platform gestures and gesture-controlled widgets. Widget-based techniques include well-researched menu techniques with custom enhancements for code editing operations, as well as

the proposed methods for source code interaction. Although individual areas have been evaluated in user studies, future work should examine the presented system as a whole. This might uncover usability issues occurring when interactions involve multiple modules.

Chapter 8

Software Architecture

Software development environments tend to evolve into applications with large code bases. The steady growth of functionality and enhancements puts emphasis on software architecture issues such as support for multiple interoperating modules or extensibility. This chapter first identifies the main areas that tend to cause considerable implementation effort and highlights the architectural differences between traditional desktop environments and touch-based environments. The second part addresses existing open-source systems, their strategies for tackling software architecture issues, and opportunities for reusing available infrastructure. The third part details principles and patterns used for implementation and describes the concrete architecture of selected sub-systems, their communication mechanisms, and technical constraints.

8.1 Introduction

IDEs tend to evolve into sophisticated software systems, often grown over several years and maintained by large development teams. Moir [BW11], for instance, stated that the open-source Eclipse project had contributions from about 1000 developers and over 170 companies (as of 2011). Although Eclipse could be considered as an exception since it has actively fostered growth through its modular structure, these numbers show the potential for IDEs to become large software systems. For smaller development teams or individuals, it seems important to early identify the areas where complexity most likely emerges. This section highlights such areas and examines the differences that arise from supporting touch-based target platforms.

8.1.1 Language Support

At a coarse level, IDEs can be divided into *language-independent* (or *language-agnostic*) and *language-specific* systems. Language-independence, in this regard, usually implies that IDEs provide *extended* support for multiple programming languages while providing at least *partial* support for other languages. Whereas partially supported features include basic text editing or syntax highlighting capabilities, extended support typically comprises features such as advanced code analysis and refactoring support. It is broadly accepted that substantial implementation effort arises from developing language-agnostic IDE components.

This matter is further complicated when IDEs not only offer extended services *within* a single programming language but also consider *cross-language* semantics. Tomassetti et al. [TVT$^+$13], for example, mention the Android platform as example for cross-language support: Android utilizes declarative XML markup in addition to the main programming language Java. When Java code references components declared in the XML file, the IDE validates the references and types to indicate potential errors. An ideal analysis infrastructure of the IDE considers references both within and across language boundaries. Furthermore, supporting multiple languages also results in increased effort regarding user interface components for editors. XML, for example, may benefit from editors that render the markup as tree of expandable and collapsible nodes; configuration files might be complemented by wizard-like UIs, and so on.

Although certain syntactic services can be developed in language-agnostic ways, deep semantic analysis usually entails redevelopment for each supported language. Parser generators like *ANTLR*[1] and meta-programming tools can assist with generating language support, but the overall complexity still increases. Language workbenches such as *MPS*[2] may decrease the implementation cost but mainly target domain-specific languages (DSLs). Moreover, the results of tools that generate IDE components as part of the language definition process are often tied to classic desktop environments or geared towards structure editing. The disparate environments of mobile touch-based devices impose constraints concerning the user interface and performance, thereby limiting the advantages of unoptimized generative approaches.

Consequently, one strategy to reduce the implementation effort for IDEs is reducing the number of supported languages, which, however, begs the question if such reduction

[1] `http://www.antlr.org/`
[2] `http://www.jetbrains.com/mps/`

also diminishes the value of an IDE. On the one hand, evidence suggests that many projects consist of artifacts written in multiple different programming languages. For instance, Tomassetti et al. note that "[...] the top 50 projects among the most active ones indexed by the Ohloh OSS directory [...] are composed, on average, by 16 distinct languages, ranging from a minimum of 3 (openSSH) to a maximum of 71 (Debian GNU/Linux)." In another examined project, *Hadoop*, they have found that "53 out 100 commits in the repository were cross language" [TVT+13].

On the other hand, language-specific tools arguably provide more useful and tailored services to developers. A reasonable compromise may exist in IDEs that cluster a subset of languages. For instance, an IDE focused on web development could provide extended support for JavaScript, HTML, and CSS files (including cross-language semantics) while supporting other artifacts only at a basic level.

8.1.2 Presentation

The differences between desktop systems and touch-based systems are most notable from a user perspective, but the underlying software architecture also reflects this difference. The user interfaces of desktop applications have traditionally been created either with native or emulated windowing toolkits that primarily render widgets and process input events from the attached hardware. Many touch-based platforms are similar to their desktop counterparts since they provide developers with sets of predefined UI components and expose mechanisms for processing events. In that regard, developing a touch-based system is similar to building a desktop system: User interface events are mapped, through a number of established communication mechanisms, to IDE actions and service methods. Touch-based interfaces, however, introduce a new dimension at the UI level. Whereas desktop systems only process keyboard events and mouse coordinates, an NUI must handle multi-touch events and recognize gestures.

To port the UI layer of existing desktop systems to touchscreens, mouse coordinates could be emulated through single-touch interaction. The approach seems to be popular with public kiosk systems that run on touch displays. However, this interaction style disregards opportunities for richer multi-touch and gesture-driven interaction and can only be seen as transitional compromise. (Also see Chapter 3 for issues associated with "touchification" of existing WIMP systems.) Since the UI layer of touch-enabled IDEs needs reconsideration, software reuse of existing systems is virtually impossible.

Despite their potential for more natural interaction, gestures also tend to cause ambiguities, lack standardized interactions, and often have low discoverability. Hence, the development of UI functionality might lead to increased solving of new interaction design issues that do not exist in desktop systems. Applications need mechanisms to interpret the added input data and handle not only single touch points but also process simultaneous points, velocities, performed patterns, and other sensor data. Moreover, predefined widgets of UI toolkits are only insufficiently suitable for code editing features. This lack of appropriate components requires more custom widgets to be developed. For central UI components like the code editor, both desktop and touch-based systems entail additional implementation effort. As demonstrated in previous chapters, however, touch-based platforms are deficient in providing efficient UI interactions for crucial features such as text selection and text entry.

Overall, it could be argued that the development of touch-based user interfaces increases rather than reduces the implementation effort.

8.1.3 Code Analysis

Modern IDEs are capable of performing most code analysis functionality in the background as the user types new code. Since the required computations do not block the user interface, developers can continue entering code while glancing at the real-time feedback. In contrast, more expensive calculations (e.g., refactoring operations) are often explicitly initiated by users.

As stated at the beginning, supporting multiple programming languages is, among other things, costly due to the added effort for re-implementing language-specific code analysis services. Even when generative techniques are utilized, the IDE has to provide the infrastructure and extension mechanisms for integrating analysis services. This section describes the components of *code analysis* (also called *code intelligence*) in more detail. Analysis services could be divided into two main areas, namely *syntactic* services and *semantic* services. This distinction has been suggested in the literature [KV10, KVKV12, ESV+13] and also becomes evident in the architecture of open-source development environments.

Syntactic Services

Syntactic services, as the name suggests, analyze the syntax of programming languages by tokenizing and parsing the source code. Since syntactic services do not depend on types and relationships between multiple files, they are sometimes bundled into standalone editor components.

Syntax highlighting, the visual coloring of tokens based on custom styles, is the prime syntactic service that almost all available text editors and IDEs supply. *Code folding* lets users expand and collapse multiple lines of code, a feature used to hide irrelevant source code. *Outline* views generate an overview (e.g., fields and methods) of the current source file and let users quickly navigate within the source file. *Bracket matching* highlights matching pairs of brackets or parentheses to give programmers hints on the structure and hierarchy of statements. *Code formatting* automatically controls the indentation and whitespace based on configured settings and thus frees programmers from manually formatting the code. Some IDEs provide, independently of concrete source control systems, *file diffs* that highlight differences between two source files side-by-side and optionally enable the merging of changes. *Smart typing* automatically inserts closing braces, tags, or punctuation during typing. Editor components can provide a degree of syntactic *code completion*, although more sophisticated completion systems exploit semantic and predictive components. *Commenting and uncommenting* of source code is a standard editor action since the format of comments varies between programming languages and developers frequently toggle the execution of entire code blocks. *Code templates and snippets* may be realized on a purely syntactic level; they insert pre-defined code blocks with optional placeholders that developers select and complete. *Search* can include both syntactic and semantic aspects but is usually implemented as a separate service.

Semantic Services

Semantic services provide deeper analysis since they exploit type information and dependencies. The absence of static type information (i.e., the use of dynamic programming languages) limits advanced code intelligence; however, type inference engines can–with accuracies ranging from precise to only “guessed”–determine the runtime types of values and thus enable useful semantic services even in dynamic environments.

Code completion displays suggestions, which are either manually invoked or shown after typing only few keystrokes. This feature has high significance in touch-based editors because it considerably saves keystrokes. *Reference highlighting* fosters program understanding by visually marking all references to a variable or field. *Semantic navigation* lets programmers jump to the definition of a variable or method, and allows for inspecting inheritance hierarchies in object-oriented programs. *Error highlighting and linting* shows erroneous or problematic source code while typing without the developer having to trigger a separate compilation step. *Help and documentation* presents abbreviated help inline or opens extended documentation to focused or selected parts of the code. *Refactoring* provides behavior preserving transformations and is frequently used by developers to improve the quality of code. *Quick fixes* and *quick assist* repair errors and perform other local code transformations.

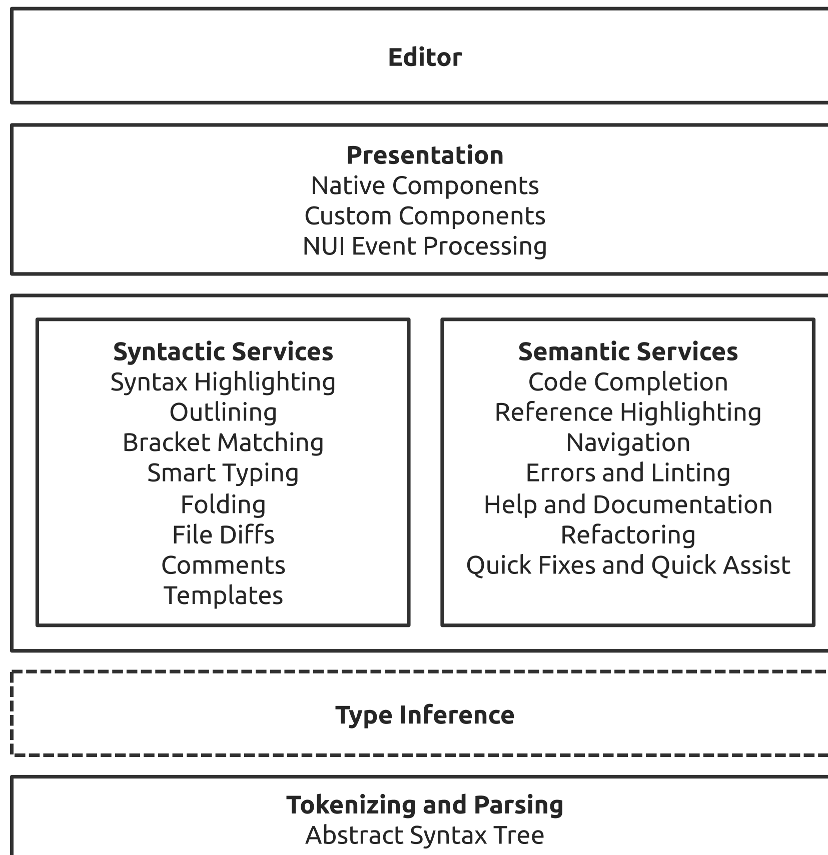


Figure 8.1: Components of an NUI code editing module.

8.1.4 Other Modules and Summary

So far, I have argued that significant IDE implementation effort arises from an NUI-compatible presentation layer and from supporting language-agnostic syntactic and semantic editor services. While touch-based systems place higher demands on the user interface, the effort for a code analysis infrastructure is–except for performance constraints on mobile platforms–similar to that of desktop systems. In addition to the UI and analysis layers, IDEs typically support other major modules like debugging, file management and version control, or building and deployment. For the rest of this chapter, I treat these functional areas as "black boxes" and concentrate on the editor, its services, and the communication mechanisms. Figure 8.1 illustrates the required components for the code editing module of an IDE.

8.2 Reference Architectures and Existing Tools

The previous overview shows that developing an IDE from scratch involves substantial work in various different areas. Thus, reusing and building upon existing solutions is essential. In this section, I first describe the architecture of the popular Eclipse IDE since it is known for its modular structure and extensibility. Eclipse has also initiated a project for a web-based IDE. This project is briefly introduced here because it demonstrates how a change of platform impacts software architecture. In addition, it shows how to manage multiple *services*, a central concept in IDE development. Second, I present reusable components for code editors and existing tools for code analysis. Without relying on such tools, IDE development would become prohibitive for individual developers or small development teams.

8.2.1 The Eclipse Project

The following section on the high-level architecture of the Eclipse (desktop) IDE is mainly based on the details given in [BW11], which reflect the state of the project as of 2011. The second section on a web-based version of Eclipse ("Orion") and its service architecture relies on the information given in the public developer guide[3] at the time of this writing.

[3] `http://wiki.eclipse.org/Orion/Documentation/Developer_Guide`

Eclipse IDE

Eclipse was first released in 2001. Its main goal was providing a modular and extensible framework that could serve as basis for developing tools for developers. The project is open-source, but employees from companies such as IBM have made a significant number of contributions. The extensible structure of Eclipse was realized through the so-called *Eclipse Component Model*, which has later been replaced by the *Eclipse Equinox* project.

The component model revolves around *plugins*, developed with the *Plugin Development Environment (PDE)*. Plugins can depend on other plugins and provide *extension points*; that is, exported interfaces that other developers can use for their own plugins. A general UI plugin, for example, provides extension points for adding menu items to a menu bar. Its menu items and mappings to dynamically instantiated classes for event processing are defined in XML configuration files. Developers can then access an in-memory *Plugin Registry* and perform queries via its API. Also, extensions are *lazily loaded* when they are first needed (e.g., when a user first clicks on a menu item), which reduces memory consumption and increases the launch time of the main application.

The presentation of Eclipse is controlled by the *Workbench* module, which manages perspectives, views, and editor windows. *Perspectives* are combinations or arrangements of different editors and views. Technically, the workbench is rendered by the *Standard Widget Toolkit (SWT)* and its complementary *JFace* framework. Eclipse achieves a consistent cross-platform look by *not* directly interacting with native OS libraries but instead relying on the abstracted SWT rendering mechanisms.

Extension points allow developers adding custom content to help and documentation, indexed by the *Apache Lucene* search engine library and served by a help server. Furthermore, Eclipse provides frameworks for building debuggers or integrating version control systems.

Eclipse 3.0 introduced, through a new project called *Equinox*, a number of replacements for the component model and the update mechanism of plugins. The already existing *OSGi*[4] specification was chosen as modularity system for dependency management and class loading. Plugins became *bundles* that could be installed, uninstalled, started, and stopped without rebooting. Update management is realized through another project

[4] `http://www.osgi.org`

called *p2*: *Installation Units* describe metadata, artifacts, and dependencies required for installation. *p2* can determine the necessary actions to set the current installation into its new state and resolves conflicts at installation time rather than at runtime.

The *Equinox* project could be regarded as concrete implementation of the OSGi specification. It also forms the basis for other projects that have evolved over the years, such as the *Rich Client Platform (RCP)*. *RCP* is a collection of smaller and more generic bundles for building Java applications and UIs that do not necessarily need IDE functionalities.

Finally, the *Eclipse 4 Application Platform (e4)* has introduced model-driven UI development, CSS styling for changing the appearance of the Eclipse application, and easier consumption of services through dependency injection. Despite these changes, developing new tools based on the Eclipse platform could be challenging. As a consequence, new *Application Services* have been introduced, aiming to simplify the interaction with core functionality.

Eclipse Orion

Eclipse Orion could be viewed as project reflecting the current trend towards IDEs that run in the web browser rather than on the desktop. Although one of the primary goals of the project is extensibility, unlike its desktop counterpart, it concentrates on supporting web-based programming languages and tooling. (It may seem obvious that a web-based IDE best supports web development technologies.) Due to Orion focusing on web technologies, the project may not need the same degree of modularity and infrastructure like the Eclipse desktop IDE. However, moving the IDE to the web poses new challenges arising from having to manage a distributed system and to work with the limited computational resources of web browsers.

In Orion, a client written in JavaScript, communicates with a server component via a REST API. Since widely deployed versions of JavaScript have lacked a single agreed upon standard for dependency management, Orion permits different external libraries for loading client-side modules. The core of the client architecture is a *Service Registry*. New services are defined in JavaScript and registered with a *Plugin Provider* that may also process supplied service properties, used to add new toolbar items and key bindings. This mechanism is similar to the extension points of the Eclipse desktop IDE. Extension points in Orion are interfaces that clients should implement. Plugins can

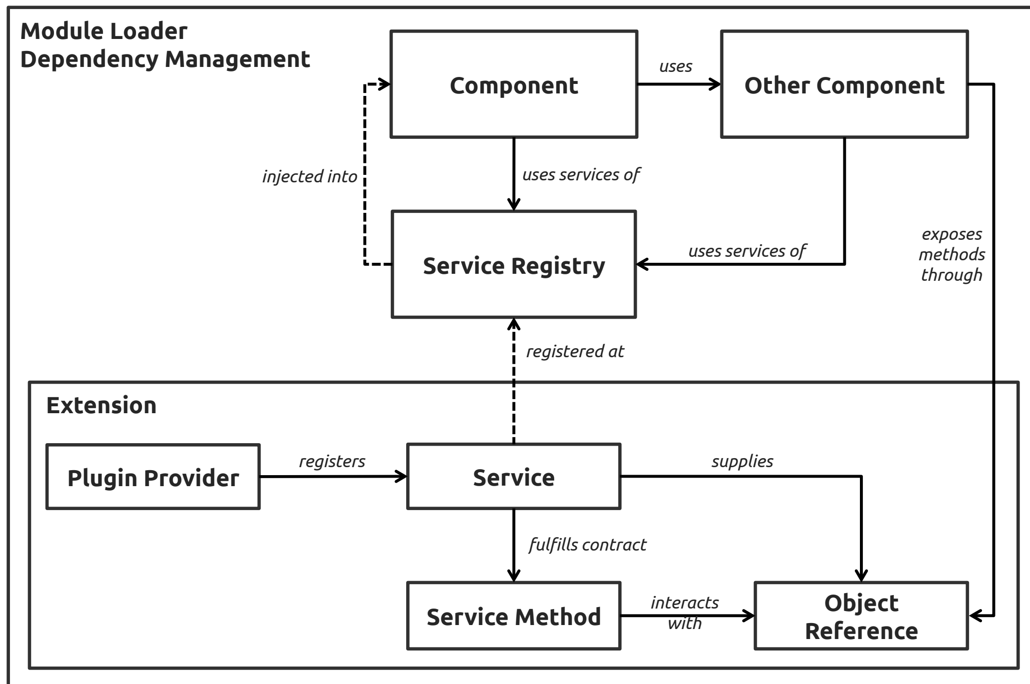


Figure 8.2: High-level interactions between components, extensions, and services in Eclipse Orion. Each component is usually defined in a separate file and loaded by a module framework that resolves all dependencies. Components can use other components, as well as the services defined in the *Service Registry*, which is injected into components. Extensions register new services with a *Plugin Provider*. Services fulfill the contract of the extension point by implementing custom functionality. Interactions with other components of the host are realized through supplied *Object References*. Components expose methods through the *Object Reference* and thus introduce indirection for consuming extensions. (This diagram is based on the description given in the Orion developer guide.)

provide multiple services and are installed via their URL from any hosting web server. Through plugins, the IDE can be extended with new functionality such as added menu commands, content assist features for different file types, or syntax highlighting rules. Services that receive configuration data before they can perform their own tasks are called *Managed Services*.

Pre-defined services include core functionality for file management, content types, preferences, and configuration data. Furthermore, Orion defines utility UI services that handle menu commands, dialogs, and messages. Another set provides extension points for customizing the editor. Developers can ask the *Editor Context*, a short-lived object reference supplied to most service providers, to return the cursor position, the

current selection, or the text of the editor window. Editing commands that transform the text in the editor or change the selection are executed on the context via a service method. Similarly, developers can contribute custom methods for calculating completion proposals, highlighting the syntax based on a declarative grammar, or outlining by applying regular expressions or inspecting the AST. Figure 8.2 illustrates the high-level interactions between components, extensions, and services.

8.2.2 Syntactic Analysis and Editor Components

Developers can take advantage of a number of open-source projects to implement the essential features of code editors. For example, JDT (Java Development Tools), a subproject of Eclipse, contributes plugins that add support for the Java programming language. The project contains all necessary functionality for building a code editing module including the syntactic and semantic services, and the corresponding views.

With the increase of web-based development tools, developers have created a number of open-source code editors. Designed as standalone components, they can be embedded into a host application. Although their main purpose is displaying syntax-highlighted code, some components expose APIs to give developers fine-grained control over the appearance and behavior of the editor. Popular projects, such as *ACE*[5] and *CodeMirror*[6], free developers from implementing the low-level details of managing text buffers and text rendering. By means of lexical analysis, the editors provide basic syntactic services and can be extended through a plugin system.

On the one hand, such components bring advantages regarding language-independence and support for web-oriented programming languages. On the other hand, standalone editors tend to blur the *view* and *model* (or *service*) layer, a conceptual and architectural separation that is regarded critical in application development. The mentioned projects achieve what is commonly referred to as "Separation of Concerns" by distinguishing between *documents* (the model) and text rendering or displaying of widgets (the view). However, with their integration into another application context, this distinction may inadvertently be removed when the component is treated as standalone module consisting of a single API.

[5] http://ace.c9.io/
[6] http://codemirror.net/

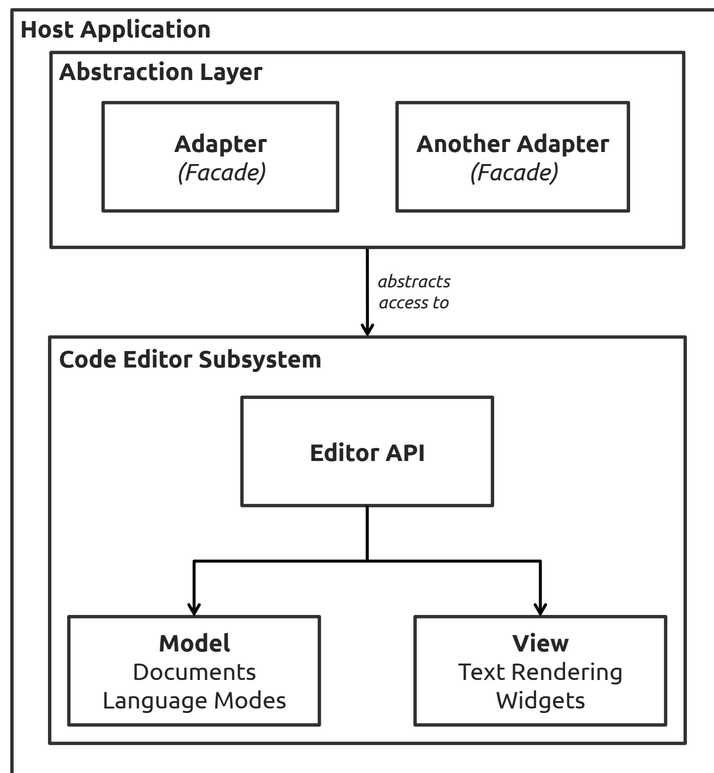


Figure 8.3: Abstraction layer for shielding the host application from changes occurring in a subsystem that utilizes pre-built code editing components. Different *Adapters* or *Facades* provide access to different functional areas of the editor subsystem in order to maintain *Separation of Concerns*.

In order to maintain the separation of concerns, the host application could add an abstraction layer. Depending on the scope, context, and implementation, abstraction layers of this type are referred to as *Adapters*, *Facades*, or *Anticorruption Layer*. *Adapters* and *Facades* are classic *GoF* software design patterns [GHJV95], whereas "Anticorruption Layer" is an established term of the *Domain Driven Design* approach [Eva04]. The primary goal of all strategies is shielding the system from changes occurring in another subsystem. Introducing this indirection allows developers to switch editor components without affecting the main application. Figure 8.3 illustrates how an abstraction layer between the host application and concrete editor subsystems could be realized.

8.2.3 Semantic and Static Analysis

Most code editor components provide only basic syntactic services based on tokenizing text. This section highlights existing tools that enable deeper semantic analysis. As for the Java programming language, the Eclipse JDT project contains the necessary building blocks. Through extension points, developers can add new functionality to the existing incremental builder and access a Java model in form of an element tree. Integrating code intelligence features typically involves manually traversing and examining the syntax tree.

Language-agnostic tooling, as previously stated, exists but loses certain benefits when integrated into other target platforms or resource-constrained environments. Particular examples include parser generators and language workbenches [KV10]. Without such tooling, developers have to directly interact with parser libraries for specific programming languages. Source code of the C language family (C, C++, Objective-C), for instance, can be analyzed by *Clang*[7], a frontend of the *LLVM* compiler infrastructure project. Since different parsers generate different AST formats, language-agnostic IDEs require additional structures for abstraction and extension. Projects such as the *Harmonia Research Project*[8] can reduce the workload since they already provide an extensible framework for language-independent assistance services. More recently, the project *srclib* has attempted to enable "polyglot code analysis" by providing "toolchains" for multiple programming languages with a "common output format" and "developer tools that consume this format"[9]. The latter is realized through installable plugins for popular code editors.

Code navigation facilities for multi-language editors could be enabled by *Ctags*[10], a popular program for scanning code and generating index files. Each line in the index file contains a tag (e.g., a class name or function name), a file path, a search pattern, and optional metadata. Code editors can parse this information to locate elements quickly within and across files.

[7] `http://clang.llvm.org/`
[8] `http://harmonia.cs.berkeley.edu/harmonia/`
[9] `https://srclib.org/`
[10] `http://ctags.sourceforge.net/`

8.3 Concrete Architecture

In this section, I describe selected patterns and communication mechanisms used to implement the system of the present work. The descriptions mainly focus on exemplary high-level architecture or areas with architectural idiosyncrasies caused by touch-based platforms.

8.3.1 Target Platform and Environment

Although the IDE has been built for the iOS platform, the principles presented here equally apply to other mobile operating systems, such as Android or Windows Mobile. The only requirement for this architecture is a platform that supports running web-based content within the native host application. The embedding is usually accomplished by *WebView* components that communicate with the host application. Since the presented system targets web-based programming languages, this design enables a high degree of reuse of existing IDE tooling for web programming. Due to performance degradation resulting from such hybrid solutions, large parts of the system including user interaction, event processing, and widget rendering are handled natively in the host application using the Objective-C programming language. Tools for cross-platform development or hybrid applications have *not* been utilized since they appear to be more appropriate for regular mobile applications built with standard components and functionalities.

8.3.2 Modules and Events

It is widely accepted that structuring a complex application into multiple loosely coupled *modules* is a desirable goal. The term "module", however, is vague. In his seminar paper "On the Criteria To Be Used in Decomposing Systems into Modules", Parnas [Par72] discusses indicators for segmenting an application into distinct parts. He considers a module as "responsibility assignment rather than a subprogram" and suggests to identify modules by "a list of difficult design decisions or design decisions which are likely to change". By means of *information hiding*, a module interface should "reveal as little as possible about its inner workings", thus reducing potential changes in other modules.

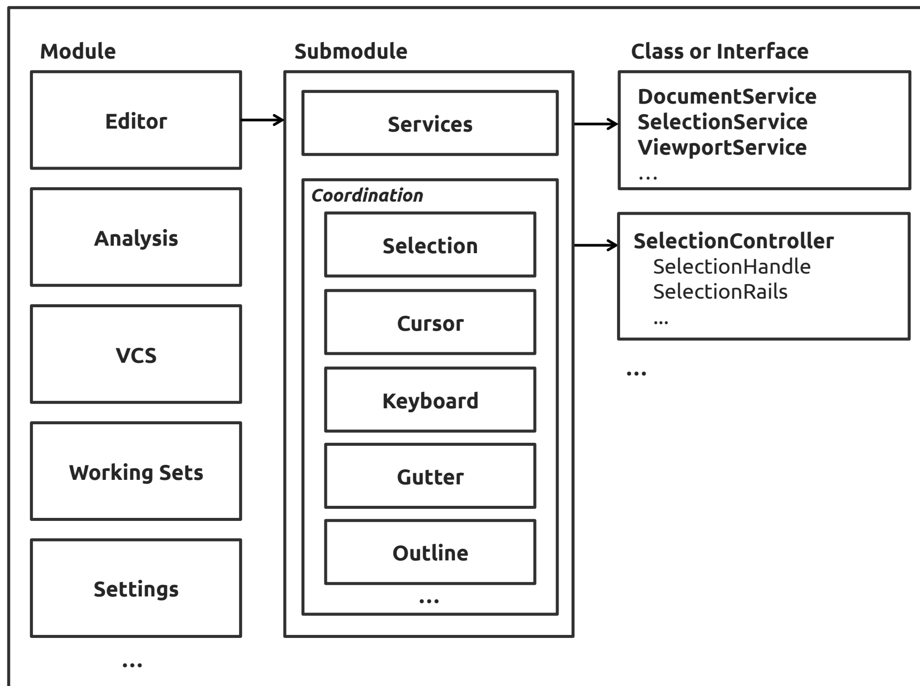


Figure 8.4: Exemplary subdivision of the application into modules.

Figure 8.4 shows how the system could be decomposed into modules, submodules, and concrete classes or interfaces. Major functional areas like the code editor or working sets form main modules that are further subdivided. The editor module, for instance, contains a set of services exposing an API for interacting with the core functionality of the editor. Furthermore, it contains view modules that group a set of classes controlling the display of UI elements.

At the class-level, a view module might internally be organized according to patterns such as *Model-View-Controller* [GHJV95] where the interface is revealed via a single controller. Parent modules may use methods of this interface to coordinate at a higher level between view modules. In order to achieve *loose coupling* between modules, communication is realized through *dependency injection* and *events* (Figure 8.5). If Module A needs to call methods of Module B, a service interface of Module B is injected into Module A. Through *polymorphism*, Module A is not bound to a concrete implementation of Module B but instead only relies on its interface definition. However, excessive use of interfaces can lead to a proliferation of dependencies in the system. Within smaller coherent modules consisting of a fixed set of known classes, interfaces

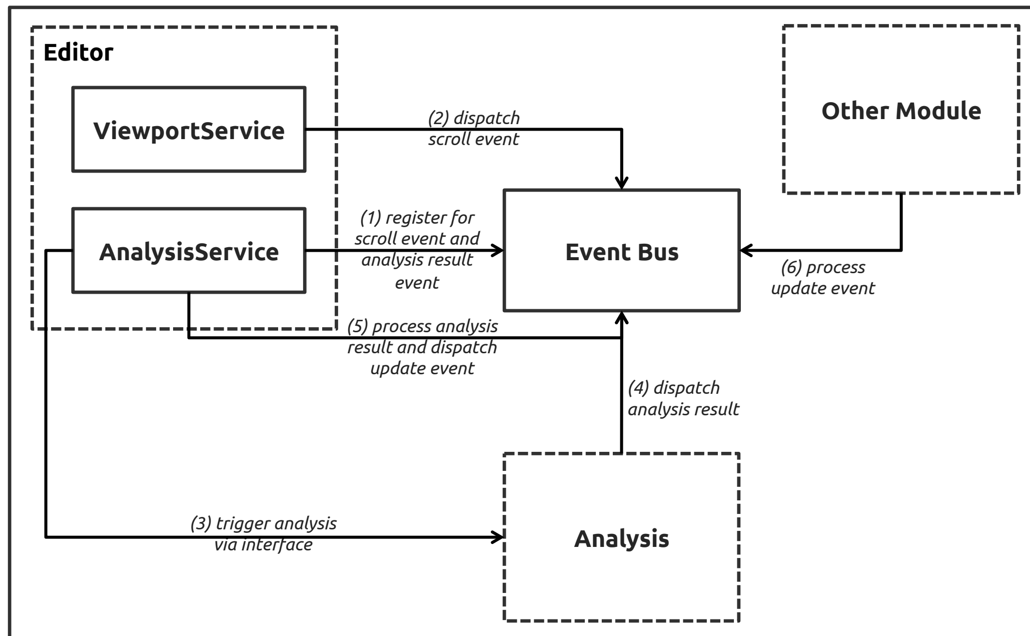


Figure 8.5: Loose coupling via interfaces and events between modules. The diagram illustrates the following exemplary interaction: When the user scrolls the editor viewport, the *ViewportService* dispatches a scroll event. The *AnalysisService* of the *Editor Module* has registered for the event and triggers a code analysis task through the interface exposed by the *Analysis Module*. When the analysis is finished, *AnalysisService* gets notified, processes the result, and dispatches an update event. Another module can register for this event and act accordingly. Although the module could directly register for the event dispatched by the *Analysis Module*, it should register for the high-level event that has been preprocessed by the coordinating *AnalysisService* of the *Editor Module*.

could be exchanged for concrete objects. Also, modules that are unlikely to change or do not modify their runtime behavior could minimize the use of abstractions.

In the above scenario, Module A still remains dependent on Module B. Coupling can be further reduced by using a central notification system: When Event B happens in Module B, Module A will be notified and react accordingly. Thus, when modules register for a system-wide event, they remain decoupled from the event source. The trade-off of this architecture is that a high volume of events could lead to inadvertent command invocations. Moreover, the debugging effort tends to increase due to difficulties in locating the actual sources of erroneous behavior in the system. In order to minimize side effects, modules could expose only few high-level events for registration.

8.3.3 Services and Core Objects

Similar to the term “module”, a “service” is a widely used but vaguely defined architectural building block. Services have primarily been discussed in the context of *Service-oriented Architecture (SOA)*.

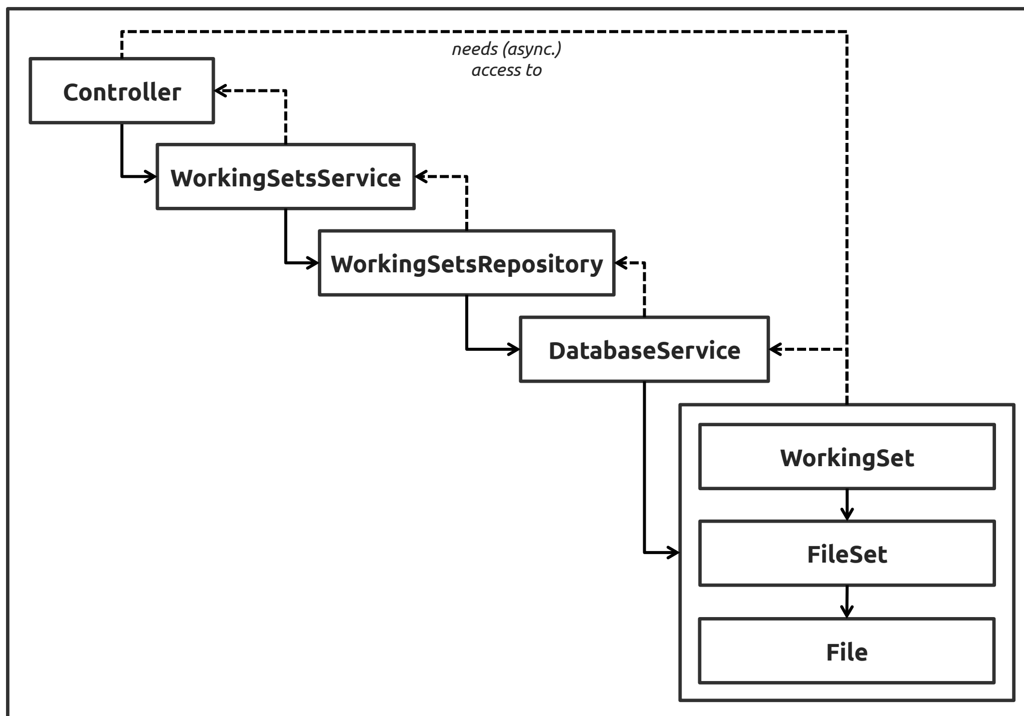


Figure 8.6: Controllers using services for interacting with core objects. For example, a controller that needs access to the *Files* of a *WorkingSet* object accesses the methods of *WorkingSetsService*. The service delegates the responsibility to a *Repository* [Eva04], decoupling the retrieval and modification of objects from their persistence mechanism. An infrastructure service handles the actual interaction with the database. The response at each step of the sequence is *asynchronous* to not block the UI during the calls.

The W3C defines “service-oriented architecture” as “set of components which can be invoked, and whose interface descriptions can be published and discovered”[11]. This description mainly refers to components that call remote procedures in a client–server environment, but services do not necessarily need to involve a remote end. Evans [Eva04], for example, more broadly distinguishes between *Application Services* and *Domain Services*. While the former execute application-specific commands and

[11] `http://www.w3.org/TR/ws-gloss/`

encapsulate technical concerns, the latter carry out operations affecting the domain model.

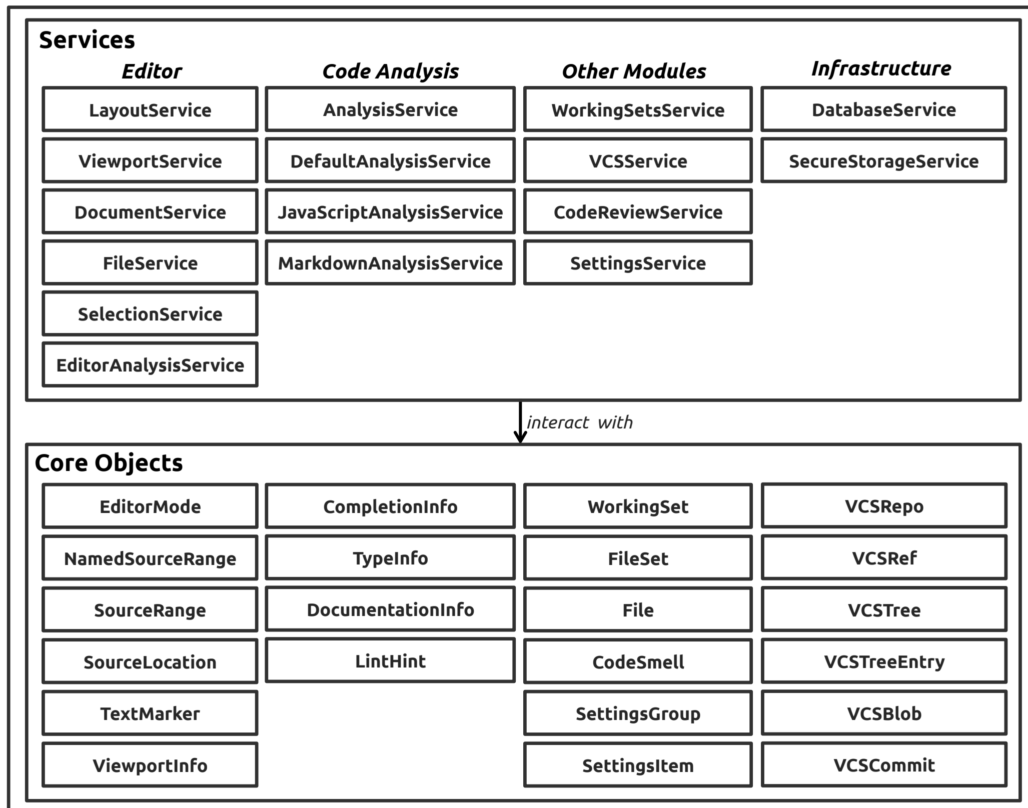


Figure 8.7: Examples of main services and core objects. The central responsibilities of a service or object are expressed through nouns. Core objects are *immutable* and can be *marshaled* into different representations to be used in both the native host application and its embedded web-based component.

As part of its *e4* platform, the Eclipse project has introduced *Application Services*, which could be regarded as *Facades* [GHJV95] for plugins that interact with the core platform or the user interface. Similarly, in this work, services represent *Facades* that alleviate clients from having to deal with implementation details. Figure 8.6 illustrates the interaction with services using the example of retrieving persisted file objects.

Figure 8.7 shows core objects that services interact with. These objects are constructed as *immutable values* to reduce failures resulting from inadvertent state changes. In addition, the objects expose mechanisms for *marshaling* (i.e., serializing) them to different representations. For instance, an object of the native application layer might

need to be converted to a JSON[12] representation when it is transferred to the embedded *WebView* component for further processing.

8.3.4 Adapters and Bridges

Adapters and *Bridges* are structural *GoF* [GHJV95] design patterns. Here, both patterns play a significant role since they encapsulate the details of the two-way hybrid interaction between the native host and the embedded *WebView*. Host adapters *wrap* the methods used to communicate with the embedded view. Services execute adapter methods without having to carry the details of *how* the adapter achieves this communication. Adapters accept or return instances of the core objects mentioned in the previous section. Similarly, the receiving end of the communication uses an adapter to process incoming method calls and to transfer objects to service methods.

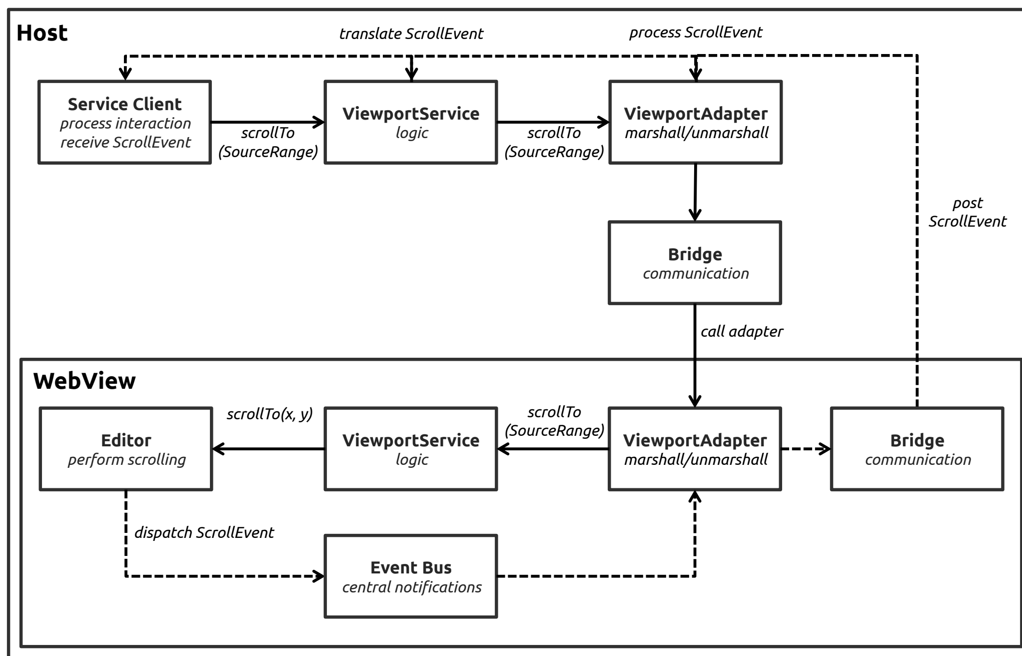


Figure 8.8: Adapters and bridges for two-way communication between the host and the embedded WebView. This example shows the high-level flow of a user-initiated event for viewport scrolling.

A *Bridge* is a related pattern of separating a concrete implementation from its interface. Gamma et al. note that "Adapter and Bridge are often used at different points in the

[12] http:/www.json.org

software lifecycle" and "The Adapter pattern makes things work after they're designed; Bridge makes them work before they are." [GHJV95]. In this work, adapters and bridges solely express different responsibilities at different levels: The bridge implements the low-level technical details of communication, whereas adapters internally use the bridge to perform their tasks at a higher level. Both levels of indirection minimize the reason for changes to service APIs. Figure 8.8 illustrates this interaction using the example of user-initiated viewport scrolling.

The services of the embedded WebView are *stateless*. When the editor changes its model or its view, those changes are communicated back to the controlling host application via events. Host adapters are the first objects to receive the events, marshal the payloads, and re-dispatch the events. Services can register for adapter events and translate them into application events that service clients may register for. (A service client could be the *Controller* of an *MVC-structured* module.)

On the one hand, this architecture introduces several layers of indirection and thereby complicates the application structure. On the other hand, it allows full reuse of prebuilt editor components. Since the editor state is synchronized, the host can reduce expensive bridged calls for querying the editor API. Through services and adapters, a single editor API is grouped into multiple logical units. The main control including event processing, handling of user interaction and gestures, and rendering of widgets remains in the host application. Because the host utilizes the native programming language of the OS, this architecture should–compared with pure web-based approaches–result in increased performance and an enhanced user experience.

8.3.5 Model-View-Controller and Commands

Model-View-Controller (MVC)[13] is an established meta-pattern, today frequently applied for developing rich user interfaces and web applications. At its core, MVC separates the presentation (view) from the data (model) while a coordinating unit (controller) connects the components. Here, modules involving views are internally structured according to MVC. Views are either native or custom-built components (widgets). Models could be core objects or more specific immutable value objects. Controllers create, update, and delete view components.

[13] `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`

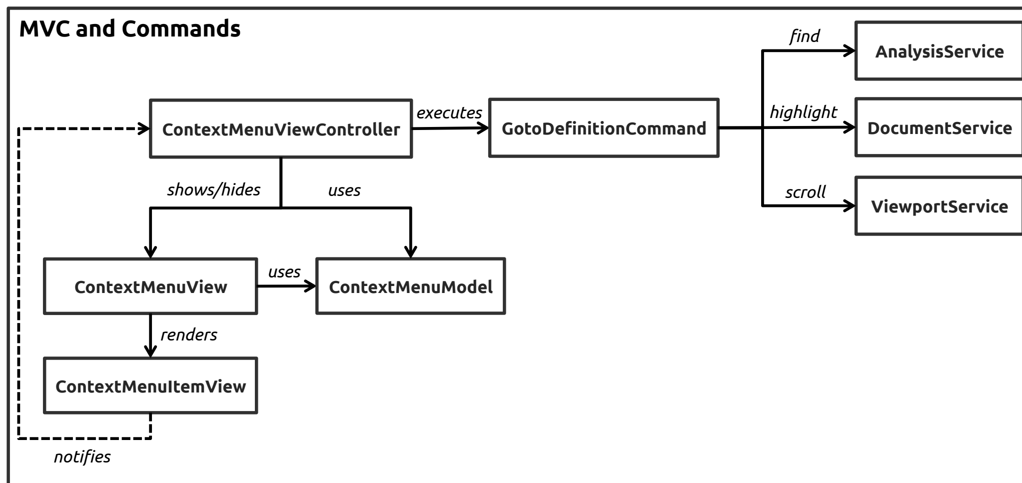


Figure 8.9: The *MVC* and *Command* patterns. The diagram shows the example of a user triggering a context menu command: To go to the definition of a variable, the user selects a context menu action. The *ViewController* handles the *View* event by extracting the *Model* data, configuring the corresponding *Command* object, and executing the command. The command finds the source range of the definition (*AnalysisService*), highlights the source range (*DocumentService*), and scrolls the editor viewport into view (*ViewportService*). After executing the command, the *ViewController* hides the *ContextMenuView*.

Controllers may also trigger stateless *Commands* [GHJV95] as part of processing events from view components. *Commands* encapsulate method invocations, thus allowing different parts of the application to execute and reuse specific tasks. Traditionally, the pattern has often been used to enable functionality such as *Undo/Redo*. *Commands* can be parameterized, and they may internally access services (Figure 8.9).

8.3.6 UI Components and Gestures

Similar to UI components of WIMP systems, NUI components could either consist of native or custom-developed elements. A UI component generally encapsulates the drawing and layout of primitive objects that compose a *widget*. Due to their interactivity, widgets are responsible for handling user-generated events. Unlike WIMP components, however, NUI components process multi-touch events and gestures. Although the OS provides event handling mechanisms, custom components frequently require extended gesture-recognition capabilities.

The UI frameworks of popular mobile platforms allow developers to extend the built-in features, for instance by creating custom *gesture recognizers*. Gesture recognizers decouple the drawing and layout of a widget from its responsibility for event handling. This separation promotes the reuse of gesture recognizers for different views. Examples of custom gesture recognizers and their relationships are shown in Figure 8.10.

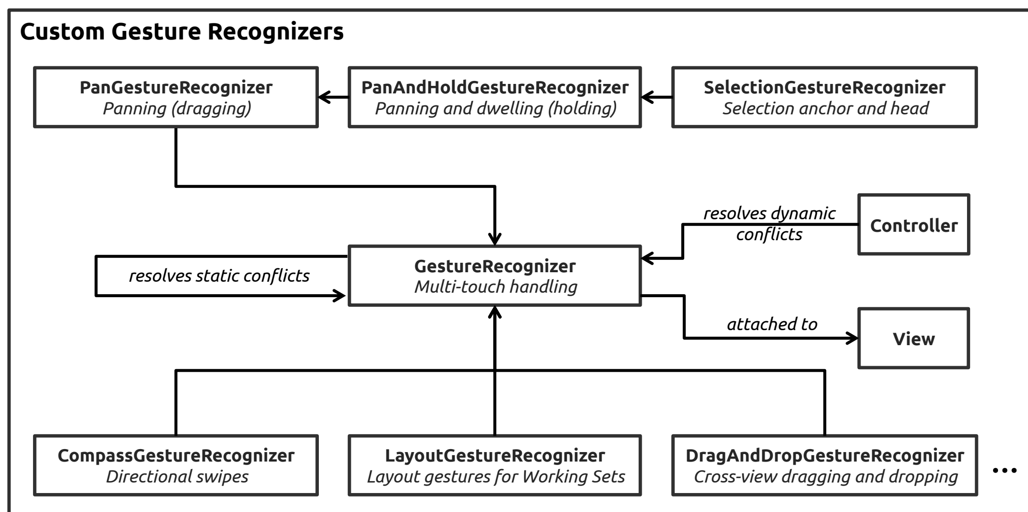


Figure 8.10: Examples of custom gesture recognizers and their (inheritance) relationships. Gesture recognizers can be associated with multiple views. Static conflicts are resolved by gesture recognizers, dynamic conflicts by controllers.

Gesture recognizers can form inheritance hierarchies. Typically, the OS framework provides a base class for multi-touch handling; that is, for accessing information about individual touches such as the location, the event phase, or timing data. In addition, most platforms define classes that detect default gestures like panning or swiping. These gestures can be extended by intercepting the event phases and implementing custom logic. Here, for example, a *PanAndHoldGestureRecognizer* extends the general pan gesture by introducing delays (dwelling) for initiating the panning and recognizing short *hold* gestures during the movement. The gesture recognizer is then attached to views that employ this interaction style (e.g., the gutter or the outline). *SelectionGestureRecognizer*, another subclass, builds upon the same basic interaction but additionally handles the selection anchor and head.

Adding multiple gesture recognizers to a view can result in *gesture conflicts*. Conflict resolution logic can either be implemented by gesture recognizers or by coordinating controllers. While the first option might be more appropriate for static and coarse

conflicts, the latter might be more adequate for dynamic or subtle conflicts. *Static* conflicts imply that one gesture should never occur simultaneously with another gesture. *Dynamic* conflicts occur when the current state (or phase) of a gesture causes another gesture to be canceled. Furthermore, the view hierarchy can be the deciding factor if a gesture should or should not be recognized (i.e., a gesture in a child view is canceled when the parent view first handles another gesture). Mobile platforms usually provide framework methods that let developers intercept touch events and prevent simultaneous recognition.

8.3.7 Concurrency and Code Analysis

While users edit code, the IDE has to update its model of the source code continually. These often computationally expensive operations should be performed in the background to keep the UI responsive at any time. Development tools, such as the previously mentioned JDT, solve this problem by *incremental builders* that track changes to project artifacts and trigger recompilation when appropriate. Integrating code analysis for dynamic programming languages usually entails generating Abstract Syntax Trees, inferring types, or finding syntactic discrepancies.

Depending on the amount of interruption for the user, background processing could be categorized into *obtrusive* and *non-obtrusive* operations. Obtrusive operations perform larger and computationally more expensive tasks like refactoring. Users may need to manually adjust parameters and wait a number of seconds until the operation is completed. Unobtrusive operations appear to be almost immediately performed as the user edits code; they do not interrupt code editing and, at most, display lightweight feedback.

Figure 8.11 shows an exemplary execution sequence for two unobtrusive code analysis operations, namely finding block statements and identifying problematic code (linting): An event, triggered when the source code is changed, reaches the *AnalysisService* that is associated with an editor. The service determines if the model needs updating and delegates the call to the language-specific *AnalysisService* of the current editor mode. An *AnalysisEngine*, running in a separate process, updates the model by performing the update through a *Worker* (thread). After the computation is finished, an event reaches the *EditorAnalysisService* that, based on the current state of the editor, decides if block statements and linting hints in the current viewport range need updating. Both operations are launched in parallel by the *AnalysisEngine*. Its responsibility is to select

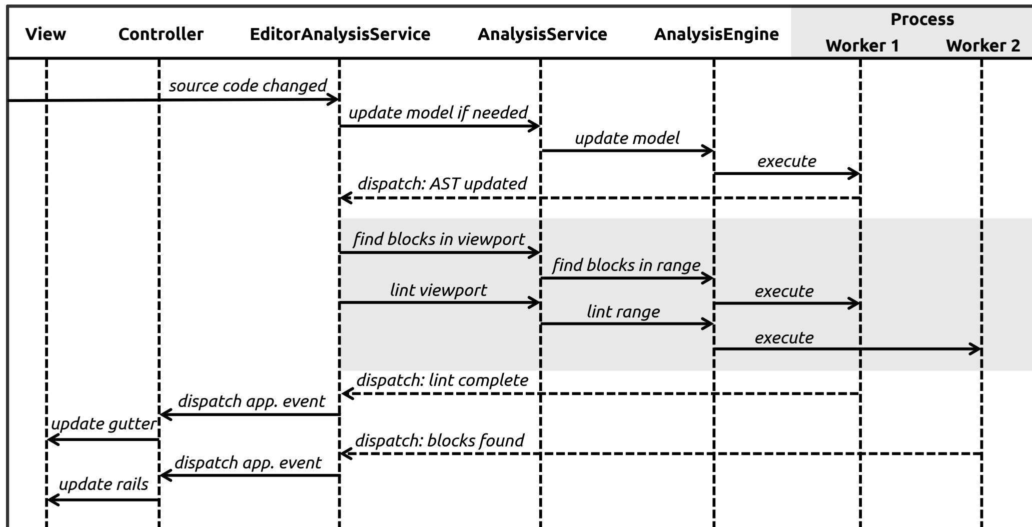


Figure 8.11: Simplified sequence diagram for concurrent code analysis operations; uses the example of updating views for block selection (selection rails) and displaying gutter hints for warnings or errors (linting). Shaded areas highlight parallel execution.

a worker to perform the operation or to launch a new worker if required. Results are asynchronously delivered back via events and ultimately handled by one or more controllers, which update their view components.

Despite the steadily increasing performance of mobile devices, mobile applications require more cautious handling of computational resources than desktop systems. Consequently, the number of parallel workers must be limited. Moreover, users might trigger some operations in rapid succession so that background workers are still busy when new work is scheduled. This issue can be mitigated by *throttling* operations; that is, calling operations at most once within a set time interval. Long-running operations should report their progress as feedback for the user, as well as display actions for canceling the operation.

8.3.8 Discussion

The previous sections have demonstrated examples of concrete object-oriented principles and patterns used to construct an IDE for mobile touch-enabled platforms. The core of the software architecture revolves around the concept of *services*. Their importance becomes evident in the architecture of major open-source IDEs (e.g., Eclipse and Eclipse Orion): Services, essentially, abstract the access to a set of responsibilities

and expose their functionality through an API to other application modules. The term "service" might also be interpreted in the broader context of object-oriented programming. Aldrich [Ald13], for example, argues that "service abstractions [...] capture the essential nature of objects" and that "the form of interoperable extension supported by service abstractions is essential to modern software". This ability, according to Aldrich, has been a critical factor for the success of object-oriented programming. IDEs with their large number of different modules appear to benefit significantly from such interoperability.

Another architectural concern stems from the need for reuse of existing IDE infrastructure and tooling. Finseth's work [Fin91] on text editing hints at the large amount of work and thought required to implement the low-level details of text editors. In addition to text editing, other modules such as code analysis functionality and artifact management require substantial implementation effort. Consequently, the architecture needs to provide mechanisms for interacting with existing libraries and doing so in ways that shield the application from changes in third-party code.

In this work, deviation from standard IDE architecture is primarily caused by the UI layer and performance constraints imposed by mobile platforms. While the latter necessarily lead to architectural simplifications and trade-offs, the NUI paradigm entails added modules for gesture recognition, more custom-developed view components, and handling of the subtleties associated with touch-centric interaction.

8.4 Conclusion

Mobile touch-enabled IDEs share architectural characteristics with desktop systems, but they also differ in many regards. I have argued that two areas in particular tend to generate complexity, namely support for language-independence and the presentation layer. Language-independence either entails manual creation of syntactic and semantic services or exploiting generative approaches. Platform constraints and different paradigms, however, may limit the usefulness of generated tooling. The interaction opportunities of NUIs translate into extended responsibilities to be handled by the presentation layer and its modules.

Furthermore, I have introduced the software architecture of Eclipse, its web-based subproject, and tooling for code analysis. To a large degree, the success of Eclipse can be attributed to its modular and extensible architecture. Although extensibility has

not been the primary goal of this work, I have emphasized the importance of services and abstractions for interacting with existing components and libraries. Concrete architectural examples have demonstrated the use of object-oriented principles and design patterns for such interaction. Only by exploiting opportunities for reuse, developers can shift their implementation efforts towards implementing a *usable* presentation and interaction layer.

Part IV

Conclusions

Chapter 9

Conclusions

In this final chapter, I first summarize the essence of previous chapters and highlight my research contributions. In the second part, I attempt to identify opportunities for future research directions and introduce two related projects that have emerged from this work. Referring to current developments, I conclude with a cautious outlook on the future of programming environments.

9.1 Summary and Contributions

The research objective of the present work could be briefly stated as "Devising techniques that improve the user interaction with textual representation of source code on touch-enabled devices".

9.1.1 Motivation

The motivation for studying touch-centric source code interaction lies in the fact that devices with touchscreens have become increasingly widespread; as a consequence, the range of applications offered extends into new domains as interaction paradigms will continue to shift from WIMP to NUI. The field of software development, however, has not yet seen the same amount of uptake in that regard as other domains. Both hardware- and software-related issues might be reasons for the slow adoption: hardware-related due to the significance of physical keyboards and sufficiently large screens for programming; software-related due to the inherent complexities of porting

sophisticated IDEs to conventional touch-enabled devices. Prior approaches have mainly employed strategies such as simplifying or constraining programming in order to circumvent the typical problems associated with code input and editing on a touchscreen.

9.1.2 Part I: Background and Prior Work

In the first part, I have introduced the context of this work by emphasizing the relationship between the process of programming and the concept of *usability*. While both the design of a programming language and the support through tooling affect usability, this work has focused on the latter. Tooling is tailored to a certain *form of representation*, a fundamental factor influencing the user interaction. Source code can be represented purely visually, purely textually, or through hybrid forms mixing textual and visual elements. Since the goal of this work has been enabling unconstrained code editing in mainstream programming languages, I have chosen textual representation as target for touch-centric enhancements.

In the second chapter of the first part, I have highlighted prior work from the intersecting research areas of *Human-Computer Interaction* and *Software Engineering*. Specifically, this has included touch- and pen-based development environments, modal and modeless text editing, novel editor interfaces, and typical IDE functionality such as code intelligence, navigation, or errors and help. Also, since IDEs provide extensive feature sets, I have shown existing solutions for efficient command invocation on touchscreens via menus, widgets, and gestures.

9.1.3 Part II: Source Code Interaction

In the second part, each of the three chapters has presented a user study examining source code interaction in different phases, namely *editing*, *selecting*, and *creating*. Editing and transforming existing code (rather than writing new code) is a frequent activity, not least due to the established practice of refactoring. The results of the first study include a user-elicited gesture set for code editing and transformation operations, and insights into the suitability of a using a pen device for input, as opposed to regular finger touches. Furthermore, the study has revealed design guidelines and limitations concerning the unambiguous integration of user-elicited gestures into a working code editor.

The second study has investigated properties of code selections, based on events gathered in realistic software development scenarios. Text selection capabilities of conventional touch-enabled platforms suffer from severe shortcomings that impede their effective use for code editing. The study results have uncovered frequently selected elements and the need for *syntax-aware* methods that take AST node boundaries into account. Guided by the findings, I have devised several gesture-driven and widget-based interaction techniques that aim at easing code selection on touchscreens, specifically: Selecting along structural boundaries through *syntax-aware* gestures, changing ranges through *selection spans* and special *handles*, accelerating line selection through *selection panning*, and selecting block statements through *selection rails*.

The third chapter has addressed the creation of new source code. The lack of a physical keyboard creates challenges for efficient input since text entry via typical virtual keyboards has notoriously been inaccurate and slow. The need for entering special characters and certain code structures has motivated the design of a *custom keyboard* optimized for code entry. In addition to an improved key layout, the keyboard includes gesture-driven triggers for code templates, code completions, and other functionality such as fast cursor movement or code deletion. I have evaluated the design of this keyboard in a user study and reported several metrics. The positive user feedback of the study has led to a revised version with an enhanced language model and touch model. The models have been improved by exploiting appropriate algorithms, and the effects have been demonstrated in simulations.

9.1.4 Part III: Design and Implementation

The third part has shown implementations of previously mentioned interaction techniques and other IDE modules, both from the perspectives of design and software architecture. First, I have described the general approach to UI and interaction design for a coherent touch-enabled IDE. I have detailed the employed strategies for disambiguating gestures and invoking commands. Following that, I have illustrated the mechanics of concrete modules that have been realized as part of this work, including file browsing, working set management, code navigation, code entry and editing, and code review.

The second chapter of this part has discussed technical aspects and software architecture. In contrast to desktop IDEs, touch-based platforms impose constraints resulting from different UI paradigms and hardware limitations. I have highlighted the architec-

tural differences and the areas that tend to cause considerable implementation effort. I have stressed the importance of reusing existing components and presented relevant reference architectures and frameworks. Using examples of selected sub-systems and their communication mechanisms, I have demonstrated how patterns and service abstractions can be exploited to realize the IDE modules.

9.2 Future Directions

In the first of the following two sections, I outline areas to which future research could be dedicated. The second section includes a brief introduction to related projects that have emerged as part of this work; these projects incorporate interaction styles that might of interest for further exploration in software development scenarios.

9.2.1 Opportunities for Further Work

Specific suggestions on how the present work could be extended include conducting further user studies, supporting hybrid representation, and enabling multi-modal interaction.

User Studies

Although I have presented three user studies investigating different aspects of working with source code, future research should continue to empirically assess the shown interaction techniques (*summative* tests). More specifically, three areas could be evaluated by complementary user studies:

- Selection techniques (Chapter 5): Although the techniques have been designed based on study results, as well as been implemented and personally tested, users might have different expectations of their mechanics. User testing would likely uncover any differences between the expected and actual selection ranges.
- Revised CEK (Chapter 6): The revised code entry keyboard realizes a modified key layout, language model, and touch model that–although already simulated–might justify a follow-up study. Also, this study could include a comparison of the CEK to a baseline keyboard so as to gain more meaningful measures of code entry speed.

- IDE UX (Chapter 7): The interface and interaction design of the supporting IDE modules have been based on findings of this work, findings of prior work, and guidelines, but could not be separately evaluated within the scope of the present work. Deploying the system to a larger group of users and collecting usage information could increase the external validity of the results.

Hybrid Representation

This work has exclusively focused on interaction techniques for textual representation of source code. However, as previously shown (Chapter 2), text can be enriched with embedded widgets. This *hybrid representation* could be realized in ways that combine the benefits of flexible textual editing and widget-based interaction. Gesture-driven interaction might be well-suited for widgets that let users visually manipulate particular elements. As a consequence, users might be able to reduce their need for more time-consuming manipulations via the keyboard. Technically, existing infrastructure such as the *IPython Kernel*[1] could be exploited to support hybrid code views.

Multi-modal Interaction

In addition to the software-based approaches of this work, hardware-based enhancements could add to the user experience and enable *multi-modal* interaction. As mentioned in Chapter 3, eye-tracking in particular appears as valuable input modality. Assuming that the required technology improves its reliability and becomes a non-invasive component of commodity hardware, I could imagine eye-tracking as meaningful extension of the programming experience. Recently, for instance, eye-tracking has been used to improve automated source code summarization [RMM+14]. Although the authors used the hardware only as an instrument for evaluation, similar approaches could be implemented in interactive forms. Furthermore, psycho-physiological measures, generated by a combination of "an eye-tracker, an electrodermal activity sensor, and an electroencephalography sensor" [FBM+14] have lately been utilized to identify problematic code in code comprehension tasks. More and more devices that measure a user's stress level and other body parameters are currently being miniaturized and offered as "wearables" for the masses. Multi-modal interaction might thus

[1] `http://ipython.org/`

become increasingly relevant for cognitively demanding scenarios such as software development.

9.2.2 Related Projects

The following two projects cover different forms of source code interaction on touch-screens. The first project proposed tabletops as suitable interactive surfaces for collaborative code reviews; the second project applied tangible interaction to exploring code smells and refactorings. The descriptions are slightly revised extracts from my publications associated with these projects [Raa11, Raa12a].

Collaborative Code Reviews on Interactive Surfaces

Chapter 7 has shown an IDE module for conducting basic code review by marking code with *review hints*. In [Raa11], I proposed the concept of a collaborative code review workflow (Figure 9.1a) that is augmented with interactive tools. This workflow is inspired by a work from Bernstein et al. [BLM$^+$10] who successfully applied a three-step process to improve the quality of crowd-sourced spelling and grammar checks in a word processing interface.

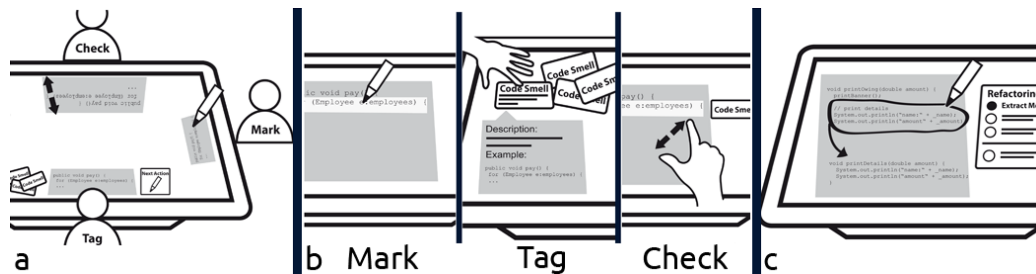


Figure 9.1: Collaborative code review workflow on an interactive tabletop.

Tailored to code reviews, the process is divided into three main steps (Figure 9.1b): 1) Reviewers mark source code that is likely to contain a bug or code smell, but they do not further specify any details; 2) Reviewers, taking only the previously marked source code into account, apply particular tags to bugs and code smells; 3) Reviewers perform quality control checks on the now marked and tagged source code; the most agreed upon defects are kept to be later addressed by refactoring tools (Figure 9.1c).

The main aims of this process are reducing variability in performance of individual reviewers, introducing a level of indirection, and fostering collaboration. At all stages, the reviewers are assisted by tools on the interactive surface. For instance, in the *mark stage*, reviewers may use a pen device to highlight certain lines in rich code views. In the *tag stage*, tangible objects representing code smells are applied to the marked lines. Finally, the *check stage* allows for efficient source code browsing through multi-touch gestures. A visual *action queue* in each reviewer's personal tabletop space highlights the next action to be performed.

The same workflow and interaction could be integrated into the tablet-based touch IDE (Chapter 7) and thus enable both synchronous and asynchronous code reviews.

Tangible Exploration of Code Smells and Refactorings

The project *CodeSmellExplorer* applies *tangible interaction* to an application that visualizes code smells and their associated refactorings [Fow99] on a tabletop (Figure 9.2). The project has been developed as an interactive exploration and learning tool after a University course that aimed at familiarizing students with principles of writing maintainable code [RFW12].

Figure 9.2: Tangible exploration of code smells and refactorings.

The tangible objects are represented by *cards*. The front of the card shows an icon and the name of a code smell, the back of the card displays a short description of the code smell and a tag that is recognized by the tabletop. *CodeSmellExplorer* lets users put their code smells on the tabletop and explore a digitally enriched display of refactorings and source code examples.

The application is based on a graph network, consisting of nodes for code smells and refactorings. Each code smell is represented by a tangible playing card that is

connected to multiple refactoring nodes as soon as the card is put onto the tabletop surface. While the code smell nodes remain at fixed position next to the tangible object, a force-based layout algorithm arranges all connected refactoring nodes so as to maintain a visually pleasing graph. In order to emphasize visual relationships, the curved connection lines are colorized to match their attached code smells. Node connections can be further inspected by tapping on the graph nodes. For instance, when the node for *Data Clumps* or *Primitive Obsession* is tapped, then the code smell, the connection, and the *Extract Method* node are highlighted. Vice versa, when the refactoring node is tapped, all connected smells and connections lines are highlighted. Refactoring nodes are weighted so that their size is adjusted according to the number of connected smells.

In addition, the tool integrates interactive challenges as *gamification* elements. For instance, when a code smell is double-tapped, users are presented with an application where code smells have to be assigned to marked ranges of the code. After completion of the challenge, a feedback visualization shows all right and wrong assignments.

9.3 The Future of Programming Environments

This work has contributed interaction techniques that aim at enhancing the interaction with source code on touchscreens. While the proposed methods are applicable *today*, future programming languages and their IDEs might introduce novel concepts that abandon the mostly text-based nature of programming. However, in light of the burden associated with supporting legacy systems for decades to come, fast transitions to new approaches appear unlikely. Since textual source code is firmly established in the industry, it will be hard for novel ideas to gain traction. Research has continually sought alternatives to textual representation but–with respect to mainstream programming–they have failed to materialize. Consequently, programming and tooling have largely remained the same over the last 40 years. Differences in modern IDEs are only evident in slight variations of similar feature sets.

At the time of this writing, observable trends in development environments include a shift towards web-based environments and live-execution models. Although the former merely constitutes a platform change, the latter impacts the interaction with source code: Users manipulate code and directly observe the effects of their changes, leading to improved feedback and a reduction of compile-run-debug cycles; essentially,

programming is brought closer to *direct manipulation*. As far as the form of representation is concerned, the programming language community has repeatedly reintroduced *reactive programming* and tools building upon the familiar spreadsheet metaphor. These approaches, however, are facing challenges in supporting large-scale software projects, and serving novices and experts alike. Although this work has primarily been motivated by the current need for improved text-centric interaction, touch-enabled devices are certainly well-suited to support alternate source code representations. It remains to be hoped that future programming affords a consistent user experience through a combination of capable programming languages, usable forms of representation, and development environments that take advantage of the interactive capabilities of modern devices. A holistic improvement of the user experience of programming might ultimately be enabled by approaches that consider interaction and tooling as an integral part of the design of a programming language.

Bibliography

[Ald13] Jonathan Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 101–116, New York, NY, USA, 2013. ACM.

[ALS10] Ahmed Sabbir Arif, Mauricio H Lopez, and Wolfgang Stuerzlinger. Two new mobile touchscreen text entry techniques. *Posters GI*, 10:22–23, 2010.

[And72] Robert H. Anderson. Programming on a tablet: A proposal for a new notation. In *Proceedings of the Symposium on Two-dimensional Man-machine Communication*, pages 113–123, New York, NY, USA, 1972. ACM.

[ARSH14] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin Schneider, and Daqing Hou. Cscc: Simple, efficient, context sensitive code completion. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, ICSME '14, 2014.

[AZ09] Caroline Appert and Shumin Zhai. Using strokes as command shortcuts: Cognitive benefits and toolkit support. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 2289–2298, New York, NY, USA, 2009. ACM.

[BBB+95] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, March 1995.

[BBMM10] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. IDE 2.0: Collective intelligence in software development. In *Proceedings of*

the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10, pages 53–58, New York, NY, USA, 2010. ACM.

[BCO+12] Xiaojun Bi, Ciprian Chelba, Tom Ouyang, Kurt Partridge, and Shumin Zhai. Bimanual gesture keyboard. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 137–146, New York, NY, USA, 2012. ACM.

[BDHM11] Andrew Bragdon, Rob DeLine, Ken Hinckley, and Meredith Ringel Morris. Code Space: Touch + air gesture hybrid interactions for supporting developer meetings. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '11, pages 212–221, New York, NY, USA, 2011. ACM.

[BG06] Andrew Begel and Susan L. Graham. An assessment of a speech-based programming environment. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 116–120, Washington, DC, USA, 2006. IEEE Computer Society.

[BHLD14] Benjamin Biegel, Julien Hoffmann, Artur Lipinski, and Stephan Diehl. U can touch this: Touchifying an IDE. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE 2014, pages 8–15, New York, NY, USA, 2014. ACM.

[BLM+10] Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. Soylent: A word processor with a crowd inside. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 313–322, New York, NY, USA, 2010. ACM.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[BNLH11] Andrew Bragdon, Eugene Nelson, Yang Li, and Ken Hinckley. Experimental analysis of touch-screen gesture designs in mobile environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 403–412, New York, NY, USA, 2011. ACM.

[Bol80] Richard A. Bolt. “Put-that-there”: Voice and gesture at the graphics interface. *SIGGRAPH Comput. Graph.*, 14(3):262–270, July 1980.

[BOZ14] Xiaojun Bi, Tom Ouyang, and Shumin Zhai. Both complete and correct?: Multi-objective optimization of touchscreen keyboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’14, pages 2297–2306, New York, NY, USA, 2014. ACM.

[BRZ+10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE ’10, pages 455–464, New York, NY, USA, 2010. ACM.

[BW11] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. Lulu Press, 2011.

[BZ13] Xiaojun Bi and Shumin Zhai. Bayesian Touch: A statistical criterion of target selection with finger touch. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST ’13, pages 51–60, New York, NY, USA, 2013. ACM.

[CLWC13] Lung-Pan Cheng, Hsiang-Sheng Liang, Che-Yang Wu, and Mike Y. Chen. iGrasp: Grasp-based adaptive keyboard for mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’13, pages 3037–3046, New York, NY, USA, 2013. ACM.

[DAB12] Edward Dillon, Monica Anderson, and Marcus Brown. Comparing mental models of novice programmers when using visual and command line environments. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE ’12, pages 142–147, New York, NY, USA, 2012. ACM.

[DBR+12] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger Canvas: Industrial experience with the code bubbles paradigm. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press.

[DLO09] Bruno Dumas, Denis Lalanne, and Sharon Oviatt. Human machine interaction. chapter Multimodal Interfaces: A Survey of Principles, Models and Frameworks, pages 3–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[DR10] Robert DeLine and Kael Rowan. Code Canvas: Zooming towards better development environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 207–210, New York, NY, USA, 2010. ACM.

[ESV+13] Sebastian Erdweg, Tijs Storm, Markus Völter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, PedroJ. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin Vlist, Guido H. Wachsmuth, and Jimi Woning. The state of the art in language workbenches. In Martin Erwig, Richard F. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling complexity in the heart of software*. Addison-Wesley, Boston, 2004.

[FBM+14] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 402–413, New York, NY, USA, 2014. ACM.

[FC12] Stephen Fitchett and Andy Cockburn. AccessRank: Predicting what users will do next. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2239–2242, New York, NY, USA, 2012. ACM.

[FHD09] Mathias Frisch, Jens Heydekorn, and Raimund Dachselt. Investigating multi-touch and pen gestures for diagram editing on interactive surfaces. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, pages 149–156, New York, NY, USA, 2009. ACM.

[FHRC14] Markus Fuchs, Markus Heckner, Felix Raab, and Wolff Christian. Monitoring students' mobile app coding behavior: Data analysis based on

IDE and browser interaction logs. In *Proceedings of the 5th IEEE Global Engineering Education Conference*, Educon '14. IEEE, 2014.

[FIM13] Vittorio Fuccella, Poika Isokoski, and Benoit Martin. Gestures and widgets: Performance in text editing on multi-touch capable mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2785–2794, New York, NY, USA, 2013. ACM.

[Fin91] Craig A. Finseth. *The Craft of Text Editing: Emacs for the Modern World*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[FKD13] G.W. French, J.R. Kennaway, and A.M. Day. Programs as visual, interactive documents. *Software: Practice and Experience*, pages 911–930, 2013.

[FLW12] Leah Findlater, Ben Lee, and Jacob Wobbrock. Beyond QWERTY: Augmenting touch screen keyboards with multi-touch gestures for non-alphanumeric input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2679–2682, New York, NY, USA, 2012. ACM.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, Reading, MA, 1999.

[FSW+14] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2491–2500, New York, NY, USA, 2014. ACM.

[FWW11] Leah Findlater, Jacob O. Wobbrock, and Daniel Wigdor. Typing on flat glass: Examining ten-finger expert typing patterns on touch surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2453–2462, New York, NY, USA, 2011. ACM.

[GCS+14] Carl Gutwin, Andy Cockburn, Joey Scarr, Sylvain Malacria, and Scott C. Olson. Faster command selection on tablets with FastTap. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2617–2626, New York, NY, USA, 2014. ACM.

[GG91] David Goldberg and Aaron Goodisman. Stylus user interfaces for manipulating text. In *Proceedings of the 4th Annual ACM Symposium on User*

Interface Software and Technology, UIST '91, pages 127–135, New York, NY, USA, 1991. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[GJM+13] Mayank Goel, Alex Jansen, Travis Mandel, Shwetak N. Patel, and Jacob O. Wobbrock. ContextType: Using hand posture information to improve mobile touch screen text entry. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2795–2798, New York, NY, USA, 2013. ACM.

[Gol83] Adele Goldberg. *Smalltalk-80: The language and its implementation*. Addison-Wesley, Reading, Mass, 1983.

[Goo87] D. Goodman. *The Complete HyperCard Handbook*. Bantam Computer Books, Birmingham, AL, 1987.

[GREW14] Hartmut Glücker, Felix Raab, Florian Echtler, and Christian Wolff. EyeDE: Gaze-enhanced software development environments. In *Proceedings of the Extended Abstracts of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI EA '14, pages 1555–1560, New York, NY, USA, 2014. ACM.

[Han03] Christopher M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[Har10] Mark Harman. Why source code analysis and manipulation will always be important. In *SCAM*, pages 7–19. IEEE Computer Society, 2010.

[HB10] Christian Holz and Patrick Baudisch. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 581–590, New York, NY, USA, 2010. ACM.

[HBKW11] John Hardy, Christopher Bull, Gerald Kotonya, and Jon Whittle. Digitally annexing desk space for software development (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 812–815, New York, NY, USA, 2011. ACM.

[HBS+12] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[HF14] Austin Z. Henley and Scott D. Fleming. The Patchworks Code Editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2511–2520, New York, NY, USA, 2014. ACM.

[HGL13] Michael E. Hansen, Robert L. Goldstone, and Andrew Lumsdaine. What makes code hard to understand? *CoRR*, abs/1304.5257, 2013.

[HLG12] Michael E. Hansen, Andrew Lumsdaine, and Robert L. Goldstone. Cognitive Architectures: A way forward for the psychology of programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 27–38, New York, NY, USA, 2012. ACM.

[HMBK10] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[HOMH12] Marc Hesenius, Carlos Dario Orozco Medina, and Dominikus Herzberg. Touching Factor: Software development on tablets. In *Proceedings of the 11th International Conference on Software Composition*, SC'12, pages 148–161, Berlin, Heidelberg, 2012. Springer-Verlag.

[HP11] Daqing Hou and David M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 233–242, Washington, DC, USA, 2011. IEEE Computer Society.

[HW85] Peter Henderson and Mark Weiser. Continuous execution: The VisiProg environment. In *Proceedings of the 8th International Conference on Soft-*

ware Engineering, ICSE ’85, pages 68–74, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[HW09] Daqing Hou and Yuejiao Wang. Analyzing the evolution of user-visible features: A case study with Eclipse. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 479–482, Sept 2009.

[HYP+10] Ken Hinckley, Koji Yatani, Michel Pahud, Nicole Coddington, Jenny Rodenhouse, Andy Wilson, Hrvoje Benko, and Bill Buxton. Pen + Touch = New Tools. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST ’10, pages 27–36, New York, NY, USA, 2010. ACM.

[IHK13] Petri Ihantola, Juha Helminen, and Ville Karavirta. How to study programming on mobile touch devices: Interactive Python code exercises. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling ’13, pages 51–58, New York, NY, USA, 2013. ACM.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[KAM05a] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE ’05, pages 126–135, New York, NY, USA, 2005. ACM.

[KAM05b] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers’ text editing. In *CHI ’05 Extended Abstracts on Human Factors in Computing Systems*, CHI EA ’05, pages 1557–1560, New York, NY, USA, 2005. ACM.

[KB93] Gordon Kurtenbach and William Buxton. The limits of expert performance using hierarchic marking menus. In *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*, CHI ’93, pages 482–487, New York, NY, USA, 1993. ACM.

[KBC+13] Per Ola Kristensson, Stephen Brewster, James Clawson, Mark Dunlop, Leah Findlater, Poika Isokoski, Benoît Martin, Antti Oulasvirta, Keith Vertanen, and Annalu Waller. Grand challenges in text entry. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 3315–3318, New York, NY, USA, 2013. ACM.

[KBCV08] Amy K. Karlson, Benjamin B. Bederson, and Jose L. Contreras-Vidal. Understanding One-Handed Use of Mobile Devices. In Joanna Lumsden, editor, *Handbook of Research on User Interface Design and Evaluation for Mobile Technology*, chapter VI, pages 86–101. Information Science Reference, 2008.

[KHHK99] Clare-Marie Karat, Christine Halverson, Daniel Horn, and John Karat. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 568–575, New York, NY, USA, 1999. ACM.

[KKD+11] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 217–224, New York, NY, USA, 2011. ACM.

[KKK+13] Jan-Peter Krämer, Thorsten Karrer, Joachim Kurz, Moritz Wittenhagen, and Jan Borchers. How tools in IDEs shape developers' navigation behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3073–3082, New York, NY, USA, 2013. ACM.

[KKKB12] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. Blaze: Supporting two-phased call graph navigation in source code. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 2195–2200, New York, NY, USA, 2012. ACM.

[KM06] Andrew J. Ko and Brad A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 387–396, New York, NY, USA, 2006. ACM.

[KMCA06] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006.

[KP05] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.

[Kri09] Per Ola Kristensson. Five challenges for intelligent text entry methods. *AI Magazine*, 30(4):85, 2009.

[KS05] Rex Bryan Kline and Ahmed Seffah. Evaluation of integrated software development environments: Challenges and results from three empirical studies. *Int. J. Hum.-Comput. Stud.*, 63(6):607–627, December 2005.

[KSL+13] Sunjun Kim, Jeongmin Son, Geehyuk Lee, Hwan Kim, and Woohun Lee. TapBoard: Making a touch screen keyboard more touchable. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 553–562, New York, NY, USA, 2013. ACM.

[KU93] Amir Ali Khwaja and Joseph E. Urban. Syntax-directed editing environments: Issues and features. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, SAC '93, pages 230–237, New York, NY, USA, 1993. ACM.

[Kur81] Thomas E. Kurtz. History of programming languages I. chapter BASIC, pages 515–537. ACM, New York, NY, USA, 1981.

[Kur93] Gordon Paul Kurtenbach. *The Design and Evaluation of Marking Menus*. PhD thesis, 1993.

[KV10] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.

[KVKV12] Lennart C.L. Kats, Richard G. Vogelij, Karl Trygve Kalleberg, and Eelco Visser. Software development environments on the web: A research agenda. In *Proceedings of the ACM International Symposium on New Ideas,*

New Paradigms, and Reflections on Programming and Software, Onward! '12, pages 99–116, New York, NY, USA, 2012. ACM.

[KZN12] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering (FSE 2012)*. Association for Computing Machinery, Inc., November 2012.

[LAV13] Luis A. Leiva, Vicent Alabau, and Enrique Vidal. Error-proof, high-performance, and context-aware gestures for interactive text edition. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 1227–1232, New York, NY, USA, 2013. ACM.

[LBM14] Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2481–2490, New York, NY, USA, 2014. ACM.

[LCJ13] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 23–32, Piscataway, NJ, USA, 2013. IEEE Press.

[LDW+13] Gierad P. Laput, Mira Dontcheva, Gregg Wilensky, Walter Chang, Aseem Agarwala, Jason Linder, and Eytan Adar. PixelTone: A multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2185–2194, New York, NY, USA, 2013. ACM.

[Leh80] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980.

[LGF10] G. Julian Lepinski, Tovi Grossman, and George Fitzmaurice. The design and evaluation of multitouch marking menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2233–2242, New York, NY, USA, 2010. ACM.

[LGYT11] Frank Chun Yat Li, Richard T. Guy, Koji Yatani, and Khai N. Truong. The 1Line Keyboard: A QWERTY layout in a single line. In *Proceedings of the*

24th Annual ACM Symposium on User Interface Software and Technology, UIST '11, pages 461–470, New York, NY, USA, 2011. ACM.

[LHKM13] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. Temporal code completion and navigation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1181–1184, Piscataway, NJ, USA, 2013. IEEE Press.

[LM11] Thomas D. LaToza and Brad A. Myers. Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 45–50, New York, NY, USA, 2011. ACM.

[LvdH11] Nicolas Lopez and André van der Hoek. The Code Orb: Supporting contextualized coding via at-a-glance views (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 824–827, New York, NY, USA, 2011. ACM.

[Mac07] I. Scott MacKenzie. *Text Entry Systems: Mobility, Accessibility, Universality*. Boston Morgan Kaufmann, Amsterdam, 2007.

[McD11] Sean McDirmid. Coding at the speed of touch. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 61–76, New York, NY, USA, 2011. ACM.

[McD13] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 53–62, New York, NY, USA, 2013. ACM.

[MFSM10] Mathew Mooty, Andrew Faulring, Jeffrey Stylos, and Brad A. Myers. Calcite: Completing code completion for constructors using crowds. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '10, pages 15–22, Washington, DC, USA, 2010. IEEE Computer Society.

[MHAB11] Emerson R. Murphy-Hill, Moin Ayazifar, and Andrew P. Black. Restructuring software with gestures. In *VL/HCC'11*, pages 165–172, 2011.

[MHB07] Emerson Murphy-Hill and Andrew P. Black. Why don't people use refactoring tools? Technical report, 1st Workshop on Refactoring Tools. TU Berlin, 2007.

[MHB08] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 421–430, New York, NY, USA, 2008. ACM.

[MHB10] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 5–14, New York, NY, USA, 2010. ACM.

[MHPB09] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[MK09] Brad Myers and Andrew Ko. The past, present and future of programming in HCI. 2009.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Softw.*, 23(4):76–83, July 2006.

[MLC09] I. Scott MacKenzie, Mauricio H. Lopez, and Steven Castelluci. Text Entry with the Apple iPhone and the Nintendo Wii. *Proceedings of CHI2009*, pages 4–9, 2009.

[MN13] Fabrice Matulic and Moira C. Norrie. Pen and touch gestural environment for document editing on interactive tabletops. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '13, pages 41–50, New York, NY, USA, 2013. ACM.

[MPK04] Brad A. Myers, John F. Pane, and Andy Ko. Natural programming languages and environments. *Commun. ACM*, 47(9):47–52, September 2004.

[MR13] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Lan-*

guages & Applications, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123, March 1990.

[NCT13] Tuan Anh Nguyen, Christoph Csallner, and Nikolai Tillmann. GROPG: A graphical on-phone debugger. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1189–1192, Piscataway, NJ, USA, 2013. IEEE Press.

[NCV+12] Stas Negara, Nicholasa Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. Using continuous change analysis to understand the practice of refactoring. Technical report, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 2012.

[NF14] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta Code, 2014.

[Nor10] Donald A. Norman. Natural user interfaces are not natural. *Interactions*, 17(3):6–10, May 2010.

[OB12] Stephen Oney and Joel Brandt. Codelets: Linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2697–2706, New York, NY, USA, 2012. ACM.

[OF13] Uran Oh and Leah Findlater. The challenges and potential of end-user gesture customization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1129–1138, New York, NY, USA, 2013. ACM.

[OHOW13] Stephen Oney, Chris Harrison, Amy Ogan, and Jason Wiese. ZoomBoard: A diminutive Qwerty soft keyboard using iterative zooming for ultra-small devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2799–2802, New York, NY, USA, 2013. ACM.

[Opd92] W. F. Opdyke. *A Program Restructuring Aid in Designing Object-Oriented Applications Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992.

[Ost97] Teresia R Ostrach. Typing speed: How fast is average: 4,000 typing scores statistically analyzed and interpreted. *Orlando, FL: Five Star Staffing*, 1997.

[OYLM12] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active Code Completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[PG06] Chris Parnin and Carsten Gorg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society.

[PGR10] Chris Parnin, Carsten Görg, and Spencer Rugaber. CodePad: Interactive spaces for maintaining concentration in programming environments. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 15–24, New York, NY, USA, 2010. ACM.

[Pik87] Rob Pike. The Text Editor Sam. *Softw. Pract. Exper.*, 17(11):813–845, November 1987.

[Pik94] Rob Pike. Acme: A user interface for programmers. In *Proceedings of the Winter 1994 USENIX Conference*, WTEC'94, pages 18–18, Berkeley, CA, USA, 1994. USENIX Association.

[Pre10] Roger Pressman. *Software Engineering: A practitioner's approach*. McGraw-Hill Higher Education, New York, 2010.

[PWM14] Frederic Pollmann, Dirk Wenig, and Rainer Malaka. HoverZoom: Making on-screen keyboards more accessible. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, pages 1261–1266, New York, NY, USA, 2014. ACM.

[Raa11] Felix Raab. Collaborative code reviews on interactive surfaces. In *Proceedings of the 29th Annual European Conference on Cognitive Ergonomics*, ECCE '11, pages 263–264, New York, NY, USA, 2011. ACM.

[Raa12a] Felix Raab. CodeSmellExplorer: Tangible exploration of code smells and refactorings. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 261–262, Sept 2012.

[Raa12b] Felix Raab. Interaktionsdesign menschzentrierter Refactoring-Tools. *Information, Wissenschaft & Praxis*, 63(5):329–334, 2012.

[RFW12] Felix Raab, Markus Fuchs, and Christian Wolff. CodingDojo: Interactive slides with real-time feedback. In Harald Reiterer and Oliver Deussen, editors, *Mensch & Computer 2012 – Workshopband: interaktiv informiert – allgegenwärtig und allumfassend!?*, pages 525–528, München, 2012. Oldenbourg Verlag.

[RL08] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[RMG+13] Quentin Roy, Sylvain Malacria, Yves Guiard, Eric Lecolinet, and James Eagan. Augmented Letters: Mnemonic gesture-based shortcuts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2325–2328, New York, NY, USA, 2013. ACM.

[RMM+14] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 390–401, New York, NY, USA, 2014. ACM.

[Rob08] Eric Roberts. *The Art & Science of Java: An introduction to computer science*. Pearson/Addison Wesley, Boston, 2008.

[RTKS13] Dimitrios Raptis, Nikolaos Tselios, Jesper Kjeldskov, and Mikael B. Skov. Does Size Matter?: Investigating the impact of mobile phone screen size on users' perceived usability, effectiveness and efficiency. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services*, MobileHCI '13, pages 127–136, New York, NY, USA, 2013. ACM.

[RWE13] Felix Raab, Christian Wolff, and Florian Echtler. RefactorPad: Editing source code on touchscreens. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 223–228, New York, NY, USA, 2013. ACM.

[Sam13] Krystian Samp. Designing graphical menus for novices and experts: Connecting design characteristics with design goals. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3159–3168, New York, NY, USA, 2013. ACM.

[SD09] Jeff Sauro and Joseph S. Dumas. Comparison of three one-question, post-task usability questionnaires. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1599–1608, New York, NY, USA, 2009. ACM.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.

[Shn83] B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, Aug 1983.

[SO13] Oliver Schoenleben and Antti Oulasvirta. Sandwich Keyboard: Fast ten-finger typing on a mobile device with adaptive touch sensing on the back side. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services*, MobileHCI '13, pages 175–178, New York, NY, USA, 2013. ACM.

[SSSS11] A. Stefik, S. Siebert, K. Slattery, and M. Stefik. Toward intuitive programming languages. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 213–214, June 2011.

[Ste98] Guy L. Steele, Jr. Growing a language. In *Addendum to the 1998 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)*, OOPSLA '98 Addendum, pages 0.01–A1, New York, NY, USA, 1998. ACM.

[Swo12] Helen Sword. *Stylish Academic Writing*. Harvard Univ Press, Cambridge, 2012.

[Tes12] Larry Tesler. A personal history of modeless text editing and cut/copy-paste. *Interactions*, 19(4):70–75, July 2012.

[TKH+14] Stuart Taylor, Cem Keskin, Otmar Hilliges, Shahram Izadi, and John Helmes. Type-hover-swipe in 96 Bytes: A motion sensing mechanical keyboard. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 1695–1704, New York, NY, USA, 2014. ACM.

[TMdHF11] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. TouchDevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 49–60, New York, NY, USA, 2011. ACM.

[TR81] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, September 1981.

[TVT+13] Federico Tomassetti, Antonio Vetró, Marco Torchiano, Markus Voelter, and Bernd Kolb. A model-based approach to language integration. In *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*, pages 76–81. IEEE, 2013.

[VCN+12] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press.

[VM10] Petcharat Viriyakattiyaporn and Gail C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 27–41, Riverton, NJ, USA, 2010. IBM Corp.

[VR09] Peter Van Roy. Programming Paradigms for Dummies: What Every Programmer Should Know. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Computer Music*. IRCAM/Delatour, 2009.

[WHM12] Julie Wagner, Stéphane Huot, and Wendy Mackay. BiTouch and BiPad: Designing bimanual interaction for hand-held tablets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2317–2326, New York, NY, USA, 2012. ACM.

[Wig10] Daniel Wigdor. *Brave NUI World: Designing natural user interfaces for touch and gesture*. Morgan Kaufmann, Burlington, MA, 2010.

[Win96] Leon E. Winslow. Programming pedagogy–a psychological overview. *SIGCSE Bull.*, 28(3):17–22, September 1996.

[WMS87] Catherine G. Wolf and Palmer Morrel-Samuels. The use of hand-drawn gestures for text editing. *Int. J. Man-Mach. Stud.*, 27(1):91–102, July 1987.

[WMW09] Jacob O. Wobbrock, Meredith Ringel Morris, and Andrew D. Wilson. User-defined gestures for surface computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1083–1092, New York, NY, USA, 2009. ACM.

[WPR+14] Daryl Weir, Henning Pohl, Simon Rogers, Keith Vertanen, and Per Ola Kristensson. Uncertain text entry on mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2307–2316, New York, NY, USA, 2014. ACM.

[WYBV12] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. SnipMatch: Using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 219–228, New York, NY, USA, 2012. ACM.

[XS06] Zhenchang Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported – an Eclipse case study. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 458–468, Sept 2006.

[YM11] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 25–30, New York, NY, USA, 2011. ACM.

[ZBAK10] Robert Zeleznik, Andrew Bragdon, Ferdi Adeputra, and Hsu-Sheng Ko. Hands-on Math: A page-based multi-touch and pen desktop for technical work and problem solving. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 17–26, New York, NY, USA, 2010. ACM.

[ZKG+09] Shumin Zhai, Per Ola Kristensson, Pengjun Gong, Michael Greiner, Shilei Allen Peng, Liang Mico Liu, and Anthony Dunnigan. Shapewriter on the iPhone: From the laboratory to the real world. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '09, pages 2667–2670, New York, NY, USA, 2009. ACM.

[ZL14] Haimo Zhang and Yang Li. GestKeyboard: Enabling gesture-based interaction on ordinary physical keyboard. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 1675–1684, New York, NY, USA, 2014. ACM.

[ZYZ+12] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.

Appendices

Appendix A

Study on Editing Source Code

On the following pages, material from the study on editing source code (Chapter 4) is presented, including:

1. The pre-study questionnaire provided to the participants before the study.
2. The list of all tasks with instructions for the experimenter.
3. The post-study questionnaire provided to the participants after the study.

Studie "RefactorPad" (1)

Einverständniserklärung

Ziel dieser Studie:
Diese Studie verfolgt den Zweck, besser zu verstehen, wie Nutzer Quellcode auf einem Touch-Screen bearbeiten würden. Die Ergebnisse sollen einen Beitrag dazu leisten, benutzerfreundliche Softwareentwicklungswerkzeuge für Geräte mit Touch-Bedienung zu entwerfen.

Ablauf:
Nach der Beantwortung des folgenden Fragebogens, werden Sie gebeten, sich für mehrere Vorgänge Interaktionsmöglichkeiten in Form von passenden Gesten zu überlegen. Dabei führen Sie jeden Vorgang in der Test-Anwendung entweder mit den Fingern oder einem Stift (Stylus) aus. Nach jeder Aufgabe werden Sie gebeten, zwei Fragen zu Ihrer Interaktion zu beantworten. Die Beantwortung dieser Fragen soll weiteren Aufschluss zur Benutzerfreundlichkeit der vorgeschlagenen Gesten geben. Bevor Sie mit der Studie anfangen, haben Sie die Möglichkeit, sich mit der Test-Anwendung vertraut zu machen und eine Beispielaufgabe zu bearbeiten.

Die Teilnahme an dieser Studie ist freiweillig. Sie können jederzeit abbrechen. Falls Sie während der Studie eine Pause möchten, geben Sie dem Testleiter einfach Bescheid.

Vertraulichkeit:
Alle Daten, die während dieser Studie gesammelt werden, sind vertraulich. Sie werden nur durch einen Code (z. B. "P1") identifiziert. Eventuelle Veröffentlichungen im Rahmen dieses Projekts enthalten keine Informationen, die Sie oder andere Teilnehmer persönlich identifizieren.

Mit der Beantwortung des Fragebogens erklären Sie sich mit diesen Informationen einverstanden.

Herzlichen Dank für Ihre Mithilfe

Felix Raab
Lehrstuhl für Medieninformatik
Universität Regensburg

Allgemeine Angaben

Ihr Alter

[ ]

Ihr Geschlecht

- ○ Männlich
- ○ Weiblich

Sind Sie Rechts- oder Linkshänder?

- ○ Rechtshänder
- ○ Linkshänder

Programmiererfahrung

Wie viele Jahre Programmiererfahrung haben Sie?

	Weniger als 1 Jahr	1 bis 2 Jahre	2 bis 5 Jahre	5 bis 10 Jahre	Mehr als 10 Jahre
	◯	◯	◯	◯	◯

Wie vertraut sind Sie mit folgenden Programmiersprachen?

	Gar nicht vertraut	Kaum vertraut	Mittelmäßig vertraut	Ziemlich vertraut	Außerordentlich vertraut
Java	◯	◯	◯	◯	◯
JavaScript	◯	◯	◯	◯	◯

Wie vertraut sind Sie mit der Entwicklungsumgebung "Eclipse"?

	Gar nicht vertraut	Kaum vertraut	Mittelmäßig vertraut	Ziemlich vertraut	Außerordentlich vertraut
	◯	◯	◯	◯	◯

Mit welchen sonstigen Programmiersprachen und / oder Entwicklungsumgebungen sind Sie mindestens "mittelmäßig vertraut"?

Sofern vorhanden, bitte kommagetrennt eingeben.

Nutzung von Touch-Screens

Wie oft nutzen Sie folgende Gerätetypen?

	Nie	Selten	Gelegentlich	Oft	Immer
Geräte mit Touch-Screens	◯	◯	◯	◯	◯
Geräte, die mit einem Stift (Pen / Stylus) bedient werden	◯	◯	◯	◯	◯

Figure A.1: Pre-study questionnaire

Task	Instruktion
Basic	
Move Caret	Bewege den Cursor in Zeile 9 nach `rentals.` *(Cursor nach `rentals.` setzen.)*
Select Identifier	Selektiere `each` in Zeile 9. *(`each` in Zeile 9 markieren.)*
Select Multiple Identifiers	Selektiere alle Vorkommen von `result.` *(`result` in Zeile 5, 36, 40, 41, 43 markieren.)*
Select Line	Selektiere Zeile 9. *(Zeile 9 selektieren.)*
Select Multiple Lines	Selektiere Zeile 2 - 5. *(Zeile 2 - 5 selektieren.)*
Select Block	Selektiere den switch-Block von Zeile 11 - 27. *(Zeile 11 - 27 selektieren.)*
Move Lines	Bewege die Zeilen 18 - 20 nach Zeile 12 - 14. *(Start: Zeilen 18 - 20 markieren. Split: Zeilen 12 - 14 markieren.)*
Duplicate Line	Dupliziere Zeile 8. *(Start: Zeile 8 markieren. Split: Zeile 8 und 9 markieren.)*
Delete Line	Lösche Zeile 29. *(Start: Zeile 29 markieren. Split: Cursor auf Zeile 29 setzen.)*
Toggle Comment	Kommentiere die if-Anweisung in Zeile 31 - 34 aus und dann wieder ein. *(Start: Zeile 31 - 34 markieren. Split: Zeile 31 - 34 markieren.)*
Copy/Paste	Kopiere die switch-Anweisung in Zeile 2 - 14 nach Zeile 18 - 30. *(Start: switch-Anweisung in Zeile 2 - 14 markieren. Split: Zeile 18 - 30 markieren.)*
Undo/Redo	Mache die letzte Aktion rückgängig und stelle sie dann wieder her. *(Split: Zeile 2 markieren.)*
Goto Method Declaration	Gehe zur Deklaration der Methode `amountFor(each)` in Zeile 11. *(Zeile 11 markieren. Scroll zu `amountFor(each)`. Zeile 50 markieren. Zurück nach oben scrollen.)*
Refactoring	
Extract Method Without Locals	Extrahiere Zeile 5 - 8 in eine neue Methode `printBanner()` und rufe diese Methode auf. *(Start: Zeile 5 - 8 markieren. Split: Zeile 18 - 23 markieren. Zeile 5 markieren.)*
Extract Method With Parameter	Extrahiere Zeile 4 - 6 in eine neue Methode `printDetails(amount)` und rufe diese Methode auf. *(Start: Zeile 4 - 6 markieren. Split: Zeile 6 - 10 markieren. Zeile 3 markieren.)*
Inline Method	Ersetze den Aufruf `moreThanFiveLateDeliveries()` in Zeile 2 durch den Body der Methode. *(Start: `moreThanFiveLateDeliveries()` in Zeile 2 markieren Zeile 6 markieren. Split: Zeile 2 markieren.)*
Inline Temp	Ersetze die Variable `basePrice` in Zeile 2 durch den Ausdruck in Zeile 1. *(Start: `basePrice` in Zeile 2 markieren. Zeile 1 markieren. Split: Rückgabewert in Zeile 1 markieren.)*
Replace Temp With Query	Extrahiere den Ausdruck in Zeile 2 in eine neue Methode `basePrice()` und ersetze alle Variablen `basePrice` durch einen Aufruf von `basePrice()`. *(Start: Zeile 2 markieren. `basePrice` in Zeile 4 und 9 selektieren. Split: Zeilen 11 - 13 markieren. Cursor in Zeile 2 setzen. `basePrice()` in Zeile 3 und 8 markieren.)*
Introduce Explaining Variable	Führe für `_quantity * _itemPrice` in Zeile 3 und 5 eine neue Variable `basePrice` ein. *(Start: `_quantity * itemPrice` in Zeile 3 und 5 markieren. Split: `basePrice` in Zeile 3, 4 und 5 markieren.)*
Rename Multiple Variables	Benenne alle Vorkommen der Variable `each` in `aRental` um. *(Start: `each` in Zeile 1, 3, 5, 6, 10, 14, 15 markieren. Split: Alle Vorkommen von `aRentals` markieren.)*

Figure A.2: Task descriptions

Studie "RefactorPad" (2)

Welche Interaktionsmethode würden Sie für die durchgeführten Aufgaben bevorzugen?

- Bedienung mit den Fingern
- Bedienung mit dem Stift (Pen / Stylus)
- Bedienung mit den Fingern und mit dem Stift (Pen / Stylus)

Welche Aktionen führen Sie sonst besonders oft in Ihrer Entwicklungsumgebung aus?

Senden

Geben Sie niemals Passwörter über Google Formulare weiter.

Powered by Google Docs

Missbrauch melden - Nutzungsbedingungen - Zusätzliche Bestimmungen

Figure A.3: Post-study questionnaire

Appendix B

Study on Selecting Source Code

On the following pages, material from the study on selecting source code (Chapter 5) is presented, including:

1. The handout provided to the students in the winter term 2012/2013.
2. The assignments sheet provided to the students in the winter term 2012/2013.
3. The handout provided to the students in the summer term 2013.
4. The assignments sheet provided to the students in the summer term 2013.

   The tasks and assignments sheets were primarily designed by Dr. Markus Heckner, who taught the Android programming courses at the University of Regensburg.

Handout zur Präsenzstudienleistung – Android WS 12 / 13

Start der Bearbeitung

1. Einrichten des Laufwerks O:
   Start > Ausführen > riotemp eingeben
   Schließen Sie den geöffneten Internet Explorer Browser.
   Jetzt steht Ihnen Laufwerk O im Windows Explorer als persönlicher Speicher zur Verfügung. Sie können den Windows Explorer über *Windows Taste + E* oder alternativ *Start > Alle Programme > Zubehör > Windows Explorer* aufrufen.

2. Kopieren der Entwicklungsumgebung
   Wechseln Sie nach *K:\PT\Medieninformatik\Kurse-MH\Android* und kopieren Sie die Datei *Android-Studienleistung.zip* in das Verzeichnis mit Ihrem NDS-Kürzel auf Laufwerk O (bereits angelegt, z.B. *O:\<IhrNDSKürzel>*).

3. Entpacken der Entwicklungsumgebung
   Klicken Sie mit der rechten Maustaste auf die kopierte Datei und wählen Sie *7Zip > Entpacken nach Android-Studienleistung* und warten Sie bis das Zip entpackt ist.
   Das Passwort lautet: androidbrain
   (Drücken Sie im Anschluss ggf. F5, falls der entpackte Ordner nicht direkt sichtbar ist).

4. Initialisieren der Entwicklungsumgebung
   Führen Sie per Doppelklick die Datei *Init.bat* im Verzeichnis *Android-Studienleistung* aus und warten Sie kurz.

5. Starten der Entwicklungsumgebung
   Starten Sie *Eclipse* über die Schnellstartleiste oder das Icon auf dem Desktop.

6. Setzen des Arbeitsbereichs
   Beim Start von *Eclipse* werden Sie nach Ihrem Arbeitsbereich (Workspace) gefragt. Setzen Sie den Workspace über den Button *Browse* auf den Ordner A*ndroid-Studienleistung/workspace*. Setzen Sie auch den Haken bei *Use this as the default...*. Beantworten Sie die Frage *Send usage statistics to Google* mit *Nein*, bestätigen Sie mit *Finish* und schließen Sie das Welcome-Tab.

7. Setzen der korrekten Java-Version
   Rufen Sie in *Eclipse* den Menübefehl *Window > Preferences* auf. Navigieren Sie in der Liste links zu *Java > Compiler* und setzen Sie den Wert im Feld *Compiler Compliance Level* auf *1.6*. Bestätigen Sie mit *OK* und beantworten Sie die Frage im Popup mit *Yes*.

8. Importieren der Projekte
   Rufen Sie den Menübefehl *File > Import* auf. Wählen Sie in der Liste *General > Existing Projects Into Workspace* und klicken Sie auf *Next*. Klicken Sie den Button *Browse* und

wählen Sie das Verzeichnis *Android-Studienleistung/projects*. Setzen Sie unten den Haken bei *Copy Projects Into Workspace*. Klicken Sie auf *Finish*. Warten Sie kurz…
Sie sehen anschließend im Package Explorer in Eclipse nun Ihre drei Aufgaben. Wählen Sie dann im Menü Project den Befehl *Clean* (*Clean All…*) und bestätigen Sie mit *OK*.

9. Anlegen der AVD für den Emulator
   9.1. Wählen Sie das erste Projekt (Quiz) und drücken Sie den *Run-Button*. Wählen Sie bei *Run As* den Menüpunkt *Android Application* und bestätigen Sie mit *OK*.
   9.2. Warten Sie kurz und beantworten Sie dann die Frage im Popup mit *Yes*.
   9.3. Wählen Sie im *Android Device Chooser* die Option *Launch a new Android Virtual Device*. Klicken Sie rechts unten auf den Button *Manager*.
   9.4. Klicken Sie im *Android Virtual Device Manager* auf den Button *New*.
   9.5. Geben Sie bei *AVD Name* folgenden Namen ein: Nexus_S. Wählen Sie als Device *Nexus S…* und bestätigen Sie mit *OK*.
   9.6. Schließen Sie den *Android Device Manager* und drücken Sie im *Android Device Chooser* auf *Refresh*. Wählen Sie dann die angelegte AVD *Nexus S…* und bestätigen Sie mit *OK*. Warten Sie kurz bis der Emulator die Anwendung startet…

Wichtige Hinweise

1. Bitte starten Sie während der Bearbeitung den Firefox-Browser nur über das Icon in der Schnellstart-Leiste oder das Desktop-Icon.
2. Starten Sie nach jeder Aufgabe Eclipse und Firefox neu!
3. Abgabe der fertig bearbeiten Studienleistung: Schalten Sie Ihren Rechner NICHT aus und melden Sie sich NICHT ab! Melden Sie sich bei uns – Wir sammeln die Aufgaben auf einem USB Datenträger ein.

Datenerhebung für ein Forschungsvorhaben

Für eine Studie im Rahmen des Projekts EVELIN zur Verbesserung der Lehre im Bereich Software-Engineering zeichnen wir die Interaktion während der Programmierung der Aufgaben auf. Diese Daten werden lediglich im Rahmen des Forschungsprojekts ausgewertet und haben keinerlei Bezug zur Bewertung Ihrer Studienleistung. Bewertet wird lediglich Ihre fertige Arbeit.

Vertraulichkeit:

Alle Daten, die während dieser Studie gesammelt werden, sind vertraulich. Sie werden nur durch einen Code (z. B. „P1“) identifiziert. Eventuelle Veröffentlichungen im Rahmen dieses Projekts enthalten keine Informationen, die Sie oder andere Teilnehmer persönlich identifizieren. Bitte geben Sie uns Bescheid, falls Sie damit nicht einverstanden sind.

Herzlichen Dank für Ihre Mithilfe und viel Erfolg!

Figure B.1: Handout in winter term 2012/13

Dr. Markus Heckner 08. Februar 2013

Präsenzstudienleistung Einführung in die Anwendungsprogrammierung (Android) WS 12 / 13

Sie haben zur Lösung der Aufgaben 120 Minuten Zeit.
Zugelassene Hilfsmittel sind alle Materialien der Vorlesung, sowie sämtliche Onlinequellen.
Öffnen Sie, bevor Sie mit den Aufgaben anfangen, immer die gegebenen Projekte und überprüfen Sie, über welche Klassen und Ressourcen das Projekt bereits verfügt.
Die Hilfe Dritter (Skype, Facebook, E-Mail, etc.) in Anspruch zu nehmen ist nicht zulässig und führt zum sofortigen Nichtbestehen der Studienleistung.

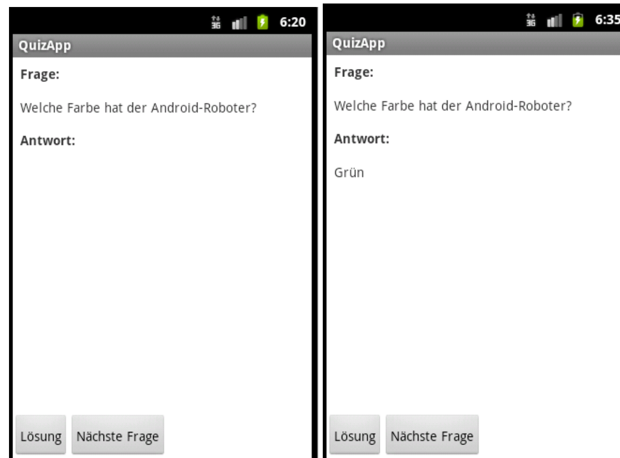
Bitte schließen Sie Eclipse nach der Bearbeitung jeder Aufgabe und starten sie es erneut zur Bearbeitung der nächsten Aufgabe.
Bitte melden Sie sich direkt nach Fertigstellung bei uns, damit wir Ihr Projekt für die Abgabe sichern können.

Aufgabe 1 QuizApp

Gegeben ist das Projekt **An_StudLstg_Presence_QuizApp_Start**. In dieser Aufgabe sollen Sie eine Quiz-App erstellen.

Die App startet mit einer Zufallsfrage, die der Spieler erraten soll. Ein Klick auf den Button *Lösung* zeigt die Lösung zur jeweiligen Quizfrage an, der Button *Nächste Frage* wechselt zur nächsten Frage. Die Auswahl der Fragen erfolgt nach dem Zufallsprinzip. Die nächste Frage erscheint zunächst wieder ohne Lösung. Und so weiter und so fort...

Die fertige App ist auf dem folgenden Screenshot dargestelt.



(a) Quiz App vor Beantwortung einer Frage (b) Quiz App mit Lösung nach Klick auf den Button Lösung

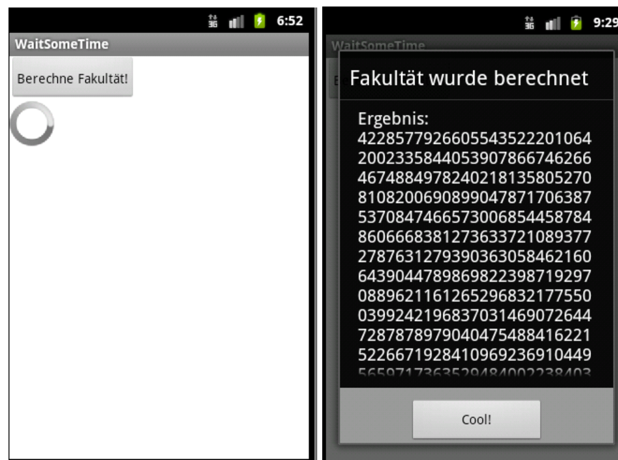
Abbildung 1: Quiz App

Achtung: Die Logik für die Applikation ist bereits implementiert. Verwenden Sie für Ihre Lösung die bestehenden Klassen `QuestionGenerator` und `QuizEntry`, die Sie **nicht** verändern müssen.

Beginnen Sie Ihre Lösung mit den in diesem Projekt gegebenen Klassen und stellen die Aufgabe gemäß der obigen Beschreibung fertig. Implementieren Sie das Layout so, wie auf dem Screenshot dargestellt (Farben können Sie vernachlässigen).

Aufgabe 2 Responsive User Interface

Gegeben ist das Projekt **An_StudLstg_Presence_Wait_Some_Time_Start**. Die App berechnet die Fakultät einer Zahl und gibt das Ergebnis wieder an den Nutzer aus. Die Berechnung der Fakultät findet in der Klasse `FacultyCalculator` statt. Hinweis: In diesem Beispiel soll immer die Fakultät von 5000 berechnet werden: Merken Sie sich diese Zahl in der Activity und übergeben Sie diese an die entsprechende Methode der Klasse `FacultyCalculator`.



(a) Fakultätsberechnung läuft, Fortschrittsanzeige wird angezeigt

(b) Fakultätsberechnung abgeschlossen, Ergebnis wird angezeigt

Abbildung 2: Fakultätsberechnung

Ergänzen Sie die folgende Funktionalität:

1. Starten Sie die Berechnung nach Klick auf den Button “Berechne Fakultät“ und geben das Ergebnis in einem AlertDialog wieder aus (vgl. Abbildung 1b). Vernachlässigen Sie in diesem ersten Schritt die Fortschrittsanzeige auf dem linken Screenshot (Abbildung 1a).
2. Das User Interface reagiert nicht, während die Berechnung läuft. Führen Sie die Berechnung im Hintergrund aus zeigen Sie während der Berechnung eine zyklische Fortschrittsanzeige an (Abbildung 1a), die sich während der Berechnung drehen muss, um anzuzeigen, dass die App aktiv “beschäftigt“ ist. Ist die Berechnung fertiggestellt, verschwindet die Fortschrittsanzeige und der Dialog aus der vorhergehenden Teilaufgabe erscheint. Achtung: Gegen Ende der Berechnung darf die Anzeige kurz anhalten, muss sich aber ansonsten laufend drehen (es ist in Ordnung, wenn die Drehung kurz stockt).

Aufgabe 3 Refactoring

Gegeben ist das Projekt **An_StudLstg_Refactoring_Start**. Die App verarbeitet eine vom Nutzer eingegebene Zeichenket-

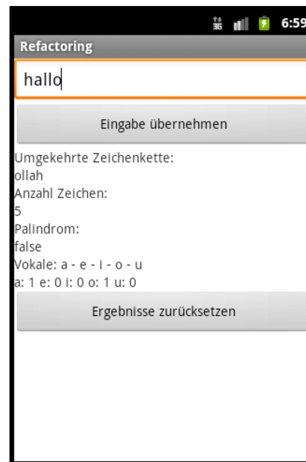


Abbildung 3: User Interface der Refactoring-App

te und gibt das Ergebnis wieder aus. Leider entspricht der Sourcecode nicht den Prinzipien guten Software-Engineerings.

Optimieren Sie den Sourcecode der Aufgabe, ohne die Funktionalität zu verändern: Passen Sie den Code dahingehend an, dass er leichter lesbar, modularer, weniger fehleranfällig und wartbarer wird. Wenden Sie die Prinzipien an, die Sie bereits aus den Veranstaltungen OOP und dem Softwareentwicklungspraktikum mit Android kennen.

Tipps:

- Ggf. können Sie auch neue Klassen erstellen und bestehende Funktionen dorthin verschieben.
- Testen Sie Ihr Programm regelmäßig, ob es auch weiterhin wie zu Beginn funktioniert.
- Die Eingabe des Emulators können Sie auf lateinische Buchstaben umstellen, in dem Sie auf die Taste links unten im Tastenfeld des Emulators drücken.

Viel Erfolg!

Figure B.2: Tasks in winter term 2012/13

Handout zur Präsenzstudienleistung – Android SS 13

Start der Bearbeitung

1. Einrichten des Laufwerks O:
   Start > Ausführen > riotemp eingeben
   Schließen Sie den geöffneten Internet Explorer Browser.
   Jetzt steht Ihnen Laufwerk O im Windows Explorer als persönlicher Speicher zur Verfügung. Sie können den Windows Explorer über *Windows Taste + E* oder alternativ *Start > Alle Programme > Zubehör > Windows Explorer* aufrufen.

2. Kopieren der Entwicklungsumgebung
   Wechseln Sie nach *K:\PT\Medieninformatik\Kurse-MH\Android* und kopieren Sie die Datei *Android-Studienleistung.zip* auf den Desktop.

3. Entpacken der Entwicklungsumgebung
   Klicken Sie mit der rechten Maustaste auf die kopierte Datei und wählen Sie *7Zip > Entpacken nach Android-Studienleistung* und warten Sie bis das Zip entpackt ist.
   Das Passwort lautet: droid4brain
   (Drücken Sie im Anschluss ggf. F5, falls der entpackte Ordner nicht direkt sichtbar ist).

4. Verschieben der Entwicklungsumgebung auf O-Laufwerk\android
   Verschieben (*Rechtsklick/Ausschneiden*) Sie den entpackten Ordner in das Verzeichnis mit Ihrem NDS-Kürzel auf Laufwerk O in den Ordner *android* (bereits angelegt, z.B. *O:\<IhrNDSKürzel>\android)*.

5. Initialisieren der Entwicklungsumgebung
   Führen Sie per Doppelklick die Datei *Init.bat* im Verzeichnis *Android-Studienleistung* aus und warten Sie kurz.

6. Starten der Entwicklungsumgebung
   Starten Sie *Eclipse* über die Schnellstartleiste oder das Icon auf dem Desktop.

7. Setzen des Arbeitsbereichs
   Beim Start von *Eclipse* werden Sie nach Ihrem Arbeitsbereich (Workspace) gefragt. Setzen Sie den Workspace über den Button *Browse* auf den Ordner *Android-Studienleistung/workspace*. Setzen Sie auch den Haken bei *Use this as the default…*. Beantworten Sie die Frage *Send usage statistics to Google* mit *Nein*, bestätigen Sie mit *Finish* und schließen Sie das Welcome-Tab.

8. Setzen der korrekten Java-Version
   Rufen Sie in *Eclipse* den Menübefehl *Window > Preferences* auf. Navigieren Sie in der Liste links zu *Java > Compiler* und setzen Sie den Wert im Feld *Compiler Compliance Level* auf *1.6*. Bestätigen Sie mit *OK* und beantworten Sie die Frage im Popup mit *Yes*.

9. Importieren der Projekte
   Rufen Sie den Menübefehl *File > Import* auf. Wählen Sie in der Liste *General > Existing Projects Into Workspace* und klicken Sie auf *Next*. Klicken Sie den Button *Browse* und wählen Sie das Verzeichnis *Android-Studienleistung/projects*. Setzen Sie unten den Haken bei *Copy Projects Into Workspace*. Klicken Sie auf *Finish*. Warten Sie kurz…
   Sie sehen anschließend im Package Explorer in Eclipse nun Ihre zwei Aufgaben. Wählen Sie dann im Menü Project den Befehl *Clean* (*Clean All…*) und bestätigen Sie mit *OK*.

10. Anlegen der AVD für den Emulator
    10.1. Wählen Sie ein Projekt und drücken Sie den *Run-Button*. Wählen Sie bei *Run As* den Menüpunkt *Android Application* und bestätigen Sie mit *OK*.
    10.2. Warten Sie kurz und beantworten Sie dann die Frage im Popup mit *Yes*.
    10.3. Wählen Sie im *Android Device Chooser* die Option *Launch a new Android Virtual Device*. Klicken Sie rechts unten auf den Button *Manager*.
    10.4. Klicken Sie im *Android Virtual Device Manager* auf den Button *New*.
    10.5. Geben Sie bei *AVD Name* folgenden Namen ein: Nexus_S. Wählen Sie als Device *Nexus S…* und als Target *Android 4.2.2* und bestätigen Sie mit *OK*.
    10.6. Schließen Sie den *Android Device Manager* und drücken Sie im *Android Device Chooser* auf *Refresh*. Wählen Sie dann die angelegte AVD *Nexus S…* und bestätigen Sie mit *OK*. Warten Sie kurz bis der Emulator die Anwendung startet…

Wichtige Hinweise

1. Bitte starten Sie während der Bearbeitung den Firefox-Browser nur über das Icon in der Schnellstart-Leiste oder das Desktop-Icon.
2. Starten Sie nach jeder Aufgabe Eclipse und Firefox neu!
3. **Abgabe** der fertig bearbeiten Studienleistung: Führen Sie die Datei *Submit.bat* im Verzeichnis *Android-Studienleistung* aus (Doppelklick) und melden Sie sich.

Datenerhebung für ein Forschungsvorhaben

Für eine Studie im Rahmen des Projekts EVELIN zur Verbesserung der Lehre im Bereich Software-Engineering zeichnen wir die Interaktion während der Programmierung der Aufgaben auf. Diese Daten werden lediglich im Rahmen des Forschungsprojekts ausgewertet und haben keinerlei Bezug zur Bewertung Ihrer Studienleistung. Bewertet wird lediglich Ihre fertige Arbeit.

Vertraulichkeit:
Alle Daten, die während dieser Studie gesammelt werden, sind vertraulich. Sie werden nur durch einen Code (z. B. „P1“) identifiziert. Eventuelle Veröffentlichungen im Rahmen dieses Projekts enthalten keine Informationen, die Sie oder andere Teilnehmer persönlich identifizieren. Bitte geben Sie uns Bescheid, falls Sie damit nicht einverstanden sind.

Herzlichen Dank für Ihre Mithilfe und viel Erfolg!

Figure B.3: Handout in summer term 2013

Dr. Markus Heckner 25. Juli 2013

Präsenzstudienleistung Einführung in die Anwendungsprogrammierung (Android) SS 13

Sie haben zur Lösung der Aufgaben 90 Minuten Zeit.
Zugelassene Hilfsmittel sind alle Materialien der Vorlesung, sowie sämtliche Onlinequellen.
Öffnen Sie, bevor Sie mit den Aufgaben anfangen, immer die gegebenen Projekte und überprüfen Sie, über welche Klassen und Ressourcen das Projekt bereits verfügt.
Die Hilfe Dritter (Skype, Facebook, E-Mail, etc.) in Anspruch zu nehmen ist nicht zulässig und führt zum sofortigen Nichtbestehen der Studienleistung.

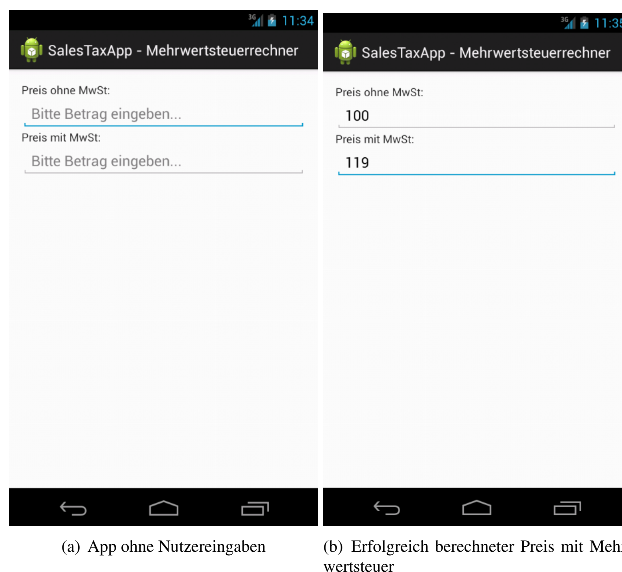
Beachten Sie die folgenden Punkte:

- Schließen Sie Eclipse nach der Bearbeitung jeder Aufgabe und starten sie es erneut zur Bearbeitung der nächsten Aufgabe.
- Melden Sie sich direkt nach Fertigstellung bei uns, damit wir Ihr Projekt für die Abgabe sichern können.
- Beginnen Sie mit Aufgabe 1, bearbeiten Sie Aufgabe 2 erst am Ende!

Aufgabe 1 Mehrwertsteuerrechner

Gegeben ist das Projekt **An_StudLstg_SalesTaxApp_Starter_Project**. In dieser Aufgabe sollen Sie eine App zur Berechnung von Preisen mit und ohne Mehrwertsteuer erstellen.
Achtung: Beginnen Sie Ihre Lösung mit den in diesem Projekt gegebenen Klassen und stellen die Aufgabe gemäß der obigen Beschreibung fertig. Implementieren Sie das Layout so, wie auf den Screenshots dargestellt. **Die App startet bereits mit einer leeren Activity, für die Sie das Layout anpassen müssen (die Layout-XML-Datei ist ebenfalls bereits im Projekt enthalten): Sie müssen keine neue Activity und keine neue Layout-XML-Datei erstellen!**

Auf dem (einzigen!) Screen der App werden dem Nutzer zwei Textfelder angezeigt, mithilfe derer sich Preise eingeben lassen. Sobald der Nutzer nach der Eingabe des Preises weiter auf dem *Soft-Keyboard* drückt (Achtung: Dies entspricht der Return-Taste auf der Tastatur Ihres Rechners, wenn Sie die App im Emulator testen), wird die Berechnung des Preises durchgeführt und das entsprechende Textfeld aktualisiert: Drückt der Nutzer Return im Textfeld des Nettopreises aktualisiert sich der Bruttopreis. Drückt der Nutzer Return im Textfeld des Bruttopreises aktualisiert sich der Nettopreis. Das User-Interface der App ist auf Abbildung 1 dargestellt.



(a) App ohne Nutzereingaben

(b) Erfolgreich berechneter Preis mit Mehrwertsteuer

Abbildung 1: App zur Berechnung der Mehrwertsteuer - Basisfunktionalität

Hinweise zur Bearbeitung:

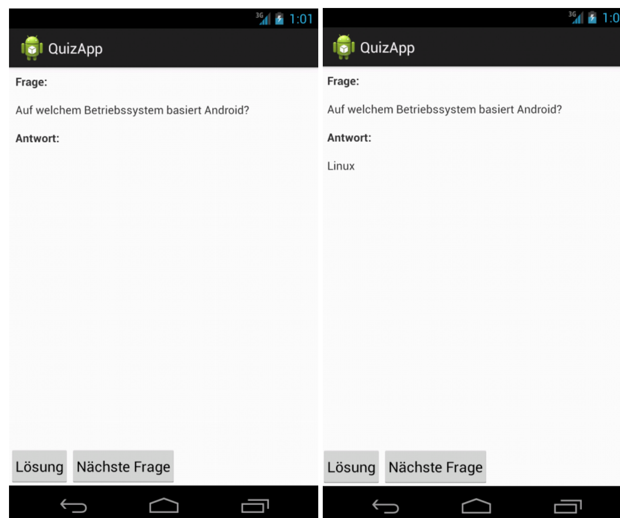
- Die Logik zur Berechnung der Steuersätze ist bereits **vollständig** implementiert. Verwenden Sie für Ihre Lösung die bestehende Klasse `SalesTaxCalculator`, die Sie **nicht** verändern müssen.
- Für das Auslösen der Berechnung kann Ihnen ein spezieller `Listener` helfen, der dann auslöst, wenn der Nutzer *weiter* bzw. *Return* auf der Tastatur des PCs, auf dem der Emulator läuft, drückt. Diesen `OnEditorActionListener` können Sie auf einem `EditText` registrieren. Nutzen Sie für die Registrierung auf dem `EditText` die Instanzmethode `setOnEditorActionListener`.
- Die Eingabe eines Textfelds können Sie lediglich auf Nummern beschränken, rufen Sie dazu die Instanzmethode `setInputType(InputType.TYPE_CLASS_NUMBER)` auf dem View auf, dessen Eingabe Sie beschränken wollen.

Aufgabe 2 Debugging einer bestehenden QuizApp

Gegeben ist das Projekt **An_StudLstg_QuizApp_Buggy**, das **3** Bugs enthält. Beheben Sie diese Bugs, sodass sich die App gemäß der folgenden Beschreibung verhält. Notieren Sie die Bugs, deren Ursache und Ihre Behebung in der im Projekt enthaltenen Datei: `BUGS AND SOLUTIONS.txt`.

Nach Klick auf `Los gehts` wechselt die App zu einer zweiten Activity, auf der das Quiz stattfindet, jedoch werden zu Beginn keine Fragen und Antworten dargestellt. Bei Klick auf `Nächste Frage` startet das Quiz mit einer Zufallsfrage, die der Spieler erraten soll. Ein Klick auf den Button *Lösung* zeigt die Lösung zur jeweiligen Quizfrage an, der Button *Nächste Frage* wechselt jetzt weiter zur nächsten Frage. Die Auswahl der Fragen erfolgt nach dem Zufallsprinzip. Die nächste Frage erscheint zunächst wieder ohne Lösung. Und so weiter und so fort...

Die fertige App ist auf dem folgenden Screenshot dargestellt.



(a) Quiz App vor Beantwortung einer Frage (b) Quiz App mit Lösung nach Klick auf den Button Lösung

Abbildung 2: Quiz App

Achtung: Legen Sie unbedingt eine Kopie des Projekts in Ihrem Workspace an, um bei Bedarf immer zum Ausgangszustand zurückkehren zu können!

Viel Erfolg!

Figure B.4: Tasks in summer term 2012/13

Appendix C

Study on Creating Source Code

On the following pages, material from the study on creating source code (Chapter 6) is presented, including:

1. The pre-study questionnaire provided to the participants before the study.
2. The program that participants had to type.
3. The extended version of the source code example.

The entry task is based on example code from Chapter 4 of *Eloquent JavaScript*[1] by Marijn Haverbeke, 2011, used under a Creative Commons Attribution 3.0 License[2].

[1] `http://eloquentjavascript.net/1st_edition/`
[2] `http://creativecommons.org/licenses/by/3.0/`

TouchCode-Study

Einverständniserklärung

Ziel dieser Studie:
Diese Studie untersucht, wie Probanden Sourcecode auf einem Tablet eingeben. Dazu werden im Anschluss an die Studie die Eigenschaften Ihrer Eingaben ausgewertet. Die Ergebnisse sollen einen Beitrag dazu leisten, die Eingabe von Sourcecode auf Geräten mit Touchscreens zu verbessern.

Ablauf:
Nach der Beantwortung des folgenden Fragebogens werden Sie mit der Testanwendung auf dem Tablet vertraut gemacht und geben dann Sourcecode von einer Papiervorlage in die Anwendung ein.

Die Teilnahme an dieser Studie ist freiwillig. Sie können jederzeit abbrechen. Falls Sie während der Studie eine Pause möchten, geben Sie dem Testleiter einfach Bescheid.

Vertraulichkeit:
Alle Daten, die während dieser Studie gesammelt werden, sind vertraulich. Sie werden nur durch einen Code (z. B. "P1") identifiziert. Eventuelle Veröffentlichungen im Rahmen dieses Projekts enthalten keine Informationen, die Sie oder andere Teilnehmer persönlich identifizieren.

Mit der Beantwortung des Fragebogens erklären Sie sich mit diesen Informationen einverstanden.

Herzlichen Dank für Ihre Mithilfe

Felix Raab
Lehrstuhl für Medieninformatik
Universität Regensburg

Ihr Alter

Ihr Geschlecht

- ◯ Männlich
- ◯ Weiblich

Sind sie Rechts- oder Linkshänder?

- ◯ Rechtshänder
- ◯ Linkshänder

Wie oft nutzen Sie folgende Gerätetypen?

Smartphones (mit Touchscreen)	○	○	○	○	○
Tablets	○	○	○	○	○

Wie viele Jahre Programmiererfahrung haben Sie?

	1 - 2 Jahre	3 - 4 Jahre	5 - 6 Jahre	Mehr als 5 Jahre	Mehr als 10 Jahre
	○	○	○	○	○

Welche Entwicklungsumgebungen oder Texteditoren nutzen Sie häufig?

Bei mehr als einer Nennung bitte kommasepariert eingeben.

Welche Programmiersprachen, Skriptsprachen oder Auszeichnungssprachen nutzen Sie häufig?

Bei mehr als einer Nennung bitte kommasepariert eingeben.

Senden

Geben Sie niemals Passwörter über Google Formulare weiter.

Bereitgestellt von

Dieser Inhalt wurde nicht von Google erstellt und wird von Google auch nicht unterstützt.

Missbrauch melden - Nutzungsbedingungen - Zusätzliche Bestimmungen

Figure C.1: Pre-study questionnaire

```
function catRecord(name) {
  return {
    name: name
  };
}

function startsWith(string, pattern) {
  return string.slice(0, pattern.length) == pattern;
}

function catNames(paragraph) {
  var colon = paragraph.indexOf(":");
  return paragraph.slice(colon + 2).split(", ");
}

function addCats(set, names) {
  for (var i = 0; i < names.length; i++) {
    set[names[i]] = catRecord(names[i]);
  }
}

function handleParagraph(paragraph) {
  if (startsWith(paragraph, "born")) {
    addCats(cats, catNames(paragraph));
  }
}

function retrieveMails() {
  return [
    "born 05/04/2006: Red Lion"
  ];
}

var cats = {};
var mailArchive = retrieveMails();
if (mailArchive.length == 0) {
  return cats;
}

for (var mail = 0; mail < mailArchive.length; mail++) {
  var paragraphs = mailArchive[mail].split("\n");
  for (var i = 0; i < paragraphs.length; i++) {
    handleParagraph(paragraphs[i]);
  }
}
return cats;
```

Figure C.2: Code entry task (shortened example code from Chapter 4 of Eloquent JavaScript by Marijn Haverbeke, 2011, used under a Creative Commons Attribution 3.0 License).

```
  function catRecord(name, birthdate, mother) {
    return {
      name: name,
      birth: birthdate,
      mother: mother
    };
  }

  function startsWith(string, pattern) {
    return string.slice(0, pattern.length) == pattern;
  }

  function catNames(paragraph) {
    var colon = paragraph.indexOf(":");
    return paragraph.slice(colon + 2).split(", ");
  }

  function extractDate(paragraph) {
    function numberAt(start, length) {
      return Number(paragraph.slice(start, start + length));
    }
    return new Date(numberAt(11, 4), numberAt(8, 2) - 1,
numberAt(5, 2));
  }

  function extractMother(paragraph) {
    var start = paragraph.indexOf("(mother ") + "(mother
".length;
    var end = paragraph.indexOf(")");
    return paragraph.slice(start, end);
  }

  function addCats(set, names, birthdate, mother) {
    for (var i = 0; i < names.length; i++) {
      set[names[i]] = catRecord(names[i], birthdate, mother);
    }
  }

  function deadCats(set, names, deathdate) {
    for (var i = 0; i < names.length; i++) {
      set[names[i]].death = deathdate;
    }
  }

  function handleParagraph(paragraph) {
    if (startsWith(paragraph, "born")) {
      addCats(cats, catNames(paragraph), extractDate(paragraph),
              extractMother(paragraph));
```

```
    } else if (startsWith(paragraph, "died")) {
      deadCats(cats, catNames(paragraph),
extractDate(paragraph));
    }
  }

  function retrieveMails() {
    return [
      "Dear nephew...",
      "etc.",
      "born 05/04/2006 (mother Lady Penelope): Red Lion"
    ];
  }

  var mailArchive = retrieveMails();
  var cats = {"Spot": catRecord("Spot", new Date(1997, 2, 5),
"unknown")};

  if (mailArchive.length == 0) {
    return cats;
  }

  for (var mail = 0; mail < mailArchive.length; mail++) {
    var paragraphs = mailArchive[mail].split("\n");
    for (var i = 0; i < paragraphs.length; i++) {
      handleParagraph(paragraphs[i]);
    }
  }

  return cats;
}

var catData = findCats();
console.log(catData);
```

Figure C.3: Extended version of the code entry task (example code from Chapter 4 of Eloquent JavaScript by Marijn Haverbeke, 2011, used under a Creative Commons Attribution 3.0 License).