Interaction-Aware Development Environments

Recording, Mining, and Leveraging IDE Interactions to Analyze and Support the Development Flow

Roberto Minelli

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera italiana (USI)

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Research Advisor

Prof. Michele Lanza

Research Co-Advisor

Dr. Andrea Mocci

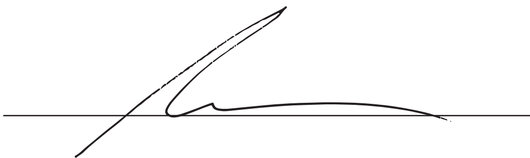
Università della Svizzera italiana

Software Institute

Dissertation Committee

Prof. Serge Demeyer	University of Antwerp, Belgium
Prof. Radu Marinescu	"Politehnica" University of Timisoara, Romania
Prof. Matthias Hauswirth	Università della Svizzera italiana (USI), Switzerland
Prof. Cesare Pautasso	Università della Svizzera italiana (USI), Switzerland

Dissertation accepted on 13 November 2017



Research Advisor

Prof. Michele Lanza

Research Co-Advisor

Dr. Andrea Mocci

Ph.D. Program Co-Director

Prof. Walter Binder

Ph.D. Program Co-Director

Prof. Olaf Schenk

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Roberto Minelli
Lugano, 13 November 2017

To those who have always been by my side.

Unconditionally.

"You can't connect the dots looking forward; you can only connect them looking backwards...

...so you have to trust that the dots will somehow connect in your future."

— Steve Jobs

Abstract

Nowadays, software development is largely carried out using Integrated Development Environments, or IDEs. An IDE is a collection of tools and facilities to support the most diverse software engineering activities, such as writing code, debugging, and program understanding. The fact that they are *integrated* enables developers to find all the tools needed for the development in the same place. Each activity is composed of many basic events, such as clicking on a menu item in the IDE, opening a new user interface to browse the source code of a method, or adding a new statement in the body of a method. While working, developers generate thousands of these interactions, that we call *fine-grained IDE interaction data.* We believe this data is a valuable source of information that can be leveraged to enable better analyses and to offer novel support to developers. However, this data is largely neglected by modern IDEs.

In this dissertation we propose the concept of *"Interaction-Aware Development Environments"*: IDEs that collect, mine, and leverage the interactions of developers to support and simplify their workflow. We formulate our thesis as follows: Interaction-Aware Development Environments enable novel and in-depth analyses of the behavior of software developers and set the ground to provide developers with effective and actionable support for their activities inside the IDE. For example, by monitoring how developers navigate source code, the IDE could suggest the program entities that are potentially relevant for a particular task.

Our research focuses on three main directions:

1. *Modeling and Persisting Interaction Data.* The first step to make IDEs aware of interaction data is to overcome its ephemeral nature. To do so we have to model this new source of data and to persist it, making it available for further use.

2. *Interpreting Interaction Data.* One of the biggest challenges of our research is making sense of the millions of interactions generated by developers. We propose several models to interpret this data, for example, by reconstructing high-level development activities from interaction histories or measure the navigation efficiency of developers.

3. *Supporting Developers with Interaction Data.* Novel IDEs can use the potential of interaction data to support software development. For example, they can identify the UI components that are potentially unnecessary for the future and suggest developers to close them, reducing the visual cluttering of the IDE.

Acknowledgements

"The journey is more important than the end or the start."
— Linkin Park

Both in everyday life and during a research doctorate, the journey is what really matters. My journey lasted 5 years, 2 months, 13 days and it was amazing. It carried me to Switzerland, Italy, the Netherlands, USA, Canada, Chile, and Japan. I devoted 18% of my lifespan to this trip. At this very moment, I am not sure I will do it again, but now it is over and it is time to take stock. For better or worse, this trip deeply changed me: I grew older, perhaps wiser, I left home, I lost touch with many people, I found a partner, and I have never stopped learning. I believe learning is the key to staying alive. Learning a new programming language, a new foreign language, or how to use the last API on the market. It makes no difference. It is important to learn, discover, and create. I spent almost every day of the last 5 years learning.

The first person who made this trip possible is my advisor, Prof. Michele Lanza. Back in 2007, being among your students for the "Programming Fundamentals 1 (PF1)" left a mark on me. Your expertise, talent, passion, and love for what you were teaching will always be unmatched. Thank you very much, I learned a lot from you. I ask you with my heart, please go back to teaching PF1. First-year students need you. It took time, but throughout the years, "Prof. Lanza" became "Michele" and REVEAL became a second family. Well, yes, I spent more time at the office than at home. I want to thank all the members of REVEAL. I learned something from each and every one of you, former and present members. Besides our very different personalities, I hope you also learned something from me. Special thanks go to Andrea Mocci, the postdoc of our research group, for all the constructive discussions over the last years.

I also want to thank the members of my dissertation committee: Prof. Serge Demeyer, Prof. Radu Marinescu, Prof. Matthias Hauswirth, and Prof. Cesare Pautasso. Thank you for the time you invested reading, understanding, and giving me insightful feedback on my dissertation. Against all expectations, I also want to thank you for the 80-minute long discussion we had during the defense. Justifying my ideas and reasoning together with you was challenging, but satisfying. I am really looking forward to meeting you again in the future.

During my Ph.D. I had the opportunity and pleasure to collaborate with Prof. Romain Robbes and Prof. Takashi Kobayashi. Thank you, I enjoyed our times together.

Thanks to Elisa Larghi, Danijela Milicevic, and Janine "Nina" Caggiano for being always responsive and for supporting me with reimbursements, travel requests, and much other paperwork. Without your help, things would not have gone so smoothly.

In spite of everything, my journey would have been impossible without the unconditional love and support of my family. Thank you Gabriella Zingali and Sergio Minelli for always believing in and being proud of me. Regardless of all my choices, all the arguments between us, and my difficult personality, I always felt your wholehearted love to push me forward. Sincere thanks also go to Paola Ghedini and Roberto Laghi, I am glad to be part of our so-called "Big Family".

Last but—needless to say—not least, there are not enough words to thank Elena Laghi for staying close to me in the most difficult time of my life. In the last five years, I struggled to bear with myself. I can not imagine how difficult it must have been for you. Together we faced many challenges, from leaving the parents' nest to organizing crazy trips around the world. I do not know what the future holds for us, but I will always be grateful for all we have been through together. Thank you, Piedino.

Roberto Minelli

Contents

Contents v

Figures xi

Tables xv

I Prologue 1

1 Introduction 3
- 1.1 Our Thesis 5
- 1.2 Contributions 5
  - 1.2.1 Modeling & Analyzing Interaction Histories 5
  - 1.2.2 Supporting Tools 5
- 1.3 Outline 6

2 IDEs and Interaction Data 9
- 2.1 From Punch Cards to Modern Development Environments 10
  - 2.1.1 From the 1980s to the Present Day 12
  - 2.1.2 Summing Up 12
- 2.2 What is Interaction Data? Why is it Important? 13
  - 2.2.1 Interactions with the IDE 14
- 2.3 Programmable looms, IDEs, and Interaction Data 15

3 The Pharo IDE 17
- 3.1 What is *Pharo*? 18
- 3.2 The Object Model of Pharo 19
  - 3.2.1 Source Code Organization 19
- 3.3 The Most Used UIs in the Pharo IDE 20
  - 3.3.1 Code Browser 20
  - 3.3.2 Workspace and Playground 20
  - 3.3.3 Inspector 21
  - 3.3.4 Debugger 22
  - 3.3.5 Search User Interfaces: Finder and Spotter 22
  - 3.3.6 Senders and Implementors Browsers 22
- 3.4 Why *Pharo*? 23
  - 3.4.1 Tab- vs. Window-based Environments 25

4 State of the Art 27
- 4.1 Recording Software Development Data 28
  - 4.1.1 The 1990s: Early Development Data 28
  - 4.1.2 The 2000s: The Interaction Data Era 28

4.2 Understanding the Behavior of Developers 29
4.2.1 Understanding Source Code Navigation 30
4.2.2 Understanding the Role of Program Comprehension 31
4.2.3 Understanding Tasks and Work Fragmentation 31
4.2.4 Leveraging Fine-Grained Source Code Changes and Biometric Data 33
4.2.5 Visualizing Software Development 33
4.3 Supporting Software Development Activities 35
4.3.1 Supporting the (Re)construction of Working Sets 35
4.3.2 Supporting Source Code Exploration and Navigation 36
4.3.3 Towards the Next Generation of IDEs 37
4.4 Privacy and Ethics 39
4.4.1 The Case of Interaction Data 39
4.4.2 Our Experience with DFlow and the *Pharo* IDE 40
4.5 Reflections 41

II Modeling, Recording, and Interpreting Interaction Data 43

5 DFlow: Our Interaction Profiler for the Pharo IDE 45
5.1 DFlow in a Nutshell 46
5.2 A Model for Interaction Data 46
5.2.1 Meta Events 46
5.2.2 User Input Events 47
5.2.3 User Interface Events 47
5.3 Evolution of DFlow 48
5.3.1 A Manual Interface to Record Development Sessions 48
5.3.2 Automatic Recording of Development Sessions 49
5.3.3 DF2low: Automatically Observing, Filtering, and Propagating Interactions 50
5.4 Reflections 52

6 A Naïve Model to Interpret Interaction Data 53
6.1 Datasets and Recording Tools 54
6.1.1 Interaction Events and Sessions Meta-Information 54
6.1.2 DFlow and Smalltalk Interaction Histories 55
6.1.3 Plog and Java Interaction Histories 56
6.2 Naïve Estimation Model 57
6.2.1 Modeling DFlow Interaction Histories 58
6.2.2 Modeling Plog Interaction Histories 61
6.2.3 Discussion: The Degrees of Freedom of the Models 63
6.3 Results 64
6.3.1 Threats to Validity 66
6.4 Reflections 67

7 Inferring High-Level Development Activities from Interaction Histories 69
7.1 The Dataset 70
7.1.1 More Than Meta Events 70
7.1.2 Facts and Figures 70
7.2 Inferring High-Level Development Activities 73
7.2.1 Events, Sprees, and Activities 73

7.2.2 Inference Model in Practice . . . 74
7.2.3 Decomposing Software Development . . . 75
7.3 How Developers Spend Their Time . . . 77
7.3.1 The Components of Program Understanding . . . 78
7.3.2 Time Spent Outside the IDE . . . 79
7.3.3 The Impact of the UI, Navigation, and Editing . . . 80
7.4 Reflections . . . 81
7.4.1 Advocatus Diaboli . . . 81
7.4.2 Wrapping Up . . . 82

8 Measuring Navigation Efficiency in the IDE 83
8.1 Source Code Navigation in the Pharo IDE . . . 84
8.1.1 Structural Source Code Navigation in *Pharo* . . . 84
8.1.2 Dataset . . . 85
8.2 Modeling Navigation Efficiency with Interaction Data . . . 85
8.3 A Naïve Model for Navigation Efficiency . . . 87
8.3.1 A More Realistic Cost Model: The Δ-cost . . . 89
8.3.2 Limitations . . . 90
8.4 A Refined Model for Navigation Efficiency . . . 92
8.4.1 Navigation Beyond the Code Browser . . . 92
8.4.2 Refining Real Navigation . . . 92
8.4.3 Refining Ideal Navigation: The UI-Aware Model . . . 93
8.4.4 Results . . . 94
8.5 Reflections . . . 94
8.5.1 Developer diversity . . . 94
8.5.2 Outliers . . . 95
8.5.3 Significance of Edited Entities . . . 95
8.5.4 Contradicting Findings? . . . 95
8.5.5 Threats to Validity . . . 95
8.6 Summing Up . . . 96

III Visual Analytics of Development Sessions 97

9 Understanding How Developers Use the User Interface of the IDE 99
9.1 Visualizing UI Usage: Principles and Proportions . . . 100
9.2 Telling Development Stories with the UI View . . . 102
9.2.1 The Dataset . . . 102
9.2.2 Development Stories . . . 103
9.3 Categorizing Developers and Development Sessions . . . 108
9.4 Reflections . . . 109

10 Visualizing the Evolution of Working Sets 111
10.1 Visualizing the Working Set . . . 112
10.1.1 What is a Working Set? . . . 112
10.1.2 Visualization Principles: Nodes, Edges, and Layout . . . 113
10.1.3 Co-Evolution of Working Set and Visualization . . . 115
10.2 Visual Analysis: Dataset and Patterns . . . 117
10.2.1 Dataset . . . 117

10.2.2 Snapshot Patterns . . . 119
10.2.3 Evolutionary Patterns . . . 124
10.3 Reflections . . . 128

11 Other Visualizations and Storytelling 129
11.1 A Catalog of Visualizations for Development Sessions . . . 130
11.1.1 Activity Forest . . . 130
11.1.2 Activity Timeline . . . 131
11.1.3 Cumulative Activity . . . 131
11.1.4 Workspace View . . . 131
11.2 Telling Visual Development Stories . . . 133
11.3 DFloWeb: Visualizing Interaction Data in the Web . . . 137
11.3.1 Visualizing Development Sessions with DFloWeb . . . 138
11.3.2 Telling Development Stories with DFloWeb . . . 139
11.4 Reflections . . . 141

IV Supporting Developers with Interaction Data 143

12 The Plague Doctor: Curing the Window Plague 145
12.1 The Plague Doctor . . . 146
12.1.1 Models and Strategies . . . 147
12.1.2 Advocatus Diaboli . . . 148
12.2 The Future of the Plague Doctor . . . 149
12.3 Summing Up . . . 150

13 Taming the User Interface of the IDE 151
13.1 IDEs and Chaotic UIs . . . 152
13.1.1 Improving Management of Working Sets . . . 152
13.1.2 Evidence from Mylyn data . . . 153
13.1.3 Beyond Tab-based IDEs . . . 153
13.1.4 Strengthening the existing evidence . . . 154
13.2 Charactering and Measuring the Chaos . . . 155
13.2.1 DFlow Dataset . . . 155
13.2.2 Modeling Chaos . . . 156
13.2.3 Wrapping Up . . . 160
13.3 Make Code, not Chaos . . . 160
13.3.1 Strategies to Tame the UI of the IDE . . . 161
13.3.2 Impact of Elision and Layout Strategies . . . 162
13.3.3 Threats to Validity . . . 164
13.3.4 Wrapping up . . . 165
13.4 Reflections . . . 165

V Epilogue 167

14 Long-Term Vision 169
14.1 Eye: The "Mother" of All Interaction Profilers . . . 170
14.1.1 A Suite of Domain-Specific Interaction Profilers . . . 171

14.1.2 All that Glitters Ain't Gold . . . 172
14.2 Recommender Systems Based on IDE Interactions . . . 172
14.3 Live and Adaptive Visualizations . . . 173
14.4 Adaptive User Interfaces . . . 174
14.5 Crowdsourced Holistic Mental Models . . . 176
14.6 Wrapping Up . . . 176

15 Conclusions **177**
15.1 Modeling, Recording, and Interpreting Interaction Data . . . 178
15.1.1 DFlow: Our Interaction Profiler for the Pharo IDE . . . 178
15.1.2 A Naïve Model to Interpret Interaction Data . . . 178
15.1.3 Inferring High-Level Development Activities from Interaction Histories . . . 178
15.1.4 Measuring Navigation Efficiency in the IDE . . . 179
15.2 Visual Analytics of Development Sessions . . . 179
15.2.1 Understanding How Developers Use the User Interface of the IDE . . . 179
15.2.2 Visualizing the Evolution of Working Sets . . . 179
15.2.3 Other Visualizations and Storytelling . . . 179
15.3 Supporting Developers with Interaction Data . . . 180
15.3.1 The Plague Doctor: Curing the Window Plague . . . 180
15.3.2 Taming the User Interface of the IDE . . . 180
15.4 Our Vision for the Future . . . 180
15.5 Closing Words . . . 181

Bibliography **183**

Online Resources **197**

VI Appendices **199**

A Blended, not Stirred: Multi-Concern Visualization of Large Software Systems **201**
A.1 The Ingredients . . . 202
A.1.1 Source Code Changes . . . 202
A.1.2 ShoreLine Reporter and Stack Traces . . . 203
A.1.3 DFlow and IDE Interaction Data . . . 203
A.2 Visualization Principles . . . 204
A.2.1 The City Metaphor: Layout and Metrics . . . 206
A.2.2 Color Harmonies and Blends . . . 206
A.2.3 Under the Hood . . . 207
A.3 Telling Development Stories with the Visualization . . . 208
A.4 Reflections . . . 215

B Gamifying Software Engineering with Interaction Data **217**
B.1 A Gamification Layer for the IDE . . . 218
B.1.1 Our vision . . . 218
B.2 Session Digest: Free Hugs for Developers . . . 218
B.3 Extending the Session Digest . . . 221
B.3.1 How Can We Evaluate a Gamification System? . . . 222
B.4 Reflections . . . 222

Figures

2.1 Interactions between the developer and different sources 13
2.2 The flow of interactions between the developer and the IDE 14

3.1 The Pharo Main Window and its UIs: (A) Playground, (B) Code Browser (or Code Editor), (C) Inspector, (D) *Spotter* Search Interface, (E) Finder UI, and (F) Debugger 18
3.2 The Code Browser of the *Pharo* IDE 20
3.3 The Workspace (3.3a) and the Playground (3.3b) of the *Pharo* IDE 21
3.4 The Inspector of the *Pharo* IDE 22
3.5 The Debugger of the *Pharo* IDE 23
3.6 Two search user interfaces of the *Pharo* IDE: Finder (3.6a) and Spotter (3.6b) 24
3.7 The Senders (3.7a) and Implementors (3.7b) Browsers of the *Pharo* IDE 24
3.8 The window-based UI of Pharo (3.8a) and the tab-based UI of Eclipse (3.8b) 25

4.1 An example of the Ownership Map exhibiting different behavioral patterns 34
4.2 Code Flow: From the source code to the visualization 34
4.3 The code_swarm visualization: an experiment in organic software visualization . . . 35
4.4 A screenshot of Code Bubbles 38

5.1 DFlow: Observing, filtering, and propagating IDE interactions 46
5.2 The model for interaction data of DFlow 47
5.3 The UI of the manual version of DFlow 49
5.4 The functioning of the manual version of DFlow 49
5.5 The functioning of the automatic version of DFlow 49
5.6 The functioning of DF2low, the most recent version of DFlow 50
5.7 Partial behavioral reflection realized with sub-method reflection 51

6.1 A development session at a glance 55
6.2 Visualizing *Java* development activities 58
6.3 Visualizing *Smalltalk* development activities 58
6.4 A raw interaction history recorded with DFlow 59
6.5 DFlow interaction history with Navigation Activities 59
6.6 DFlow interaction history with Navigation and Editing Activities 59
6.7 DFlow interaction history with Navigation, Editing, and Inspecting Activities . . . 60
6.8 DFlow interaction history with all the Activities 60
6.9 The case of editing after inspection 60
6.10 A raw interaction history recorded with Plog 62
6.11 Plog interaction history with Navigation Activities 62
6.12 Plog interaction history with Navigation and Editing Activities 62
6.13 Plog interaction history with Navigation and Editing Activities 63
6.14 Development activities for all sessions 64

7.1 Sprees and Activities from fine-grained interaction histories 74

7.2 How do developers spend their time? . . . 77

8.1 The Code Browser: The main UI to Navigate and modify code in the *Pharo* IDE . . 84
8.2 Navigating source code in the *Pharo* Code Browser: (a) Selecting a Package, (b) a Class, (c) a Protocol, and (d) a Method . . . 87
8.3 Sorting entities to minimize the Δ-cost . . . 89
8.4 Navigation Efficiency per developer [Δ-cost + sequence] . . . 91
8.5 The *Pharo* Debugger UI . . . 91
8.6 The Senders UI (a) and the Implementors UI (b) for the method `size` . . . 92
8.7 Navigation Efficiency considering Navigations outside the Code Browser . . . 93
8.8 Results – UI-Aware Navigation Efficiency per developer . . . 94

9.1 Principles and proportions of the visualization . . . 100
9.2 Visualizing the same session (a) before and (b) after the removal of pauses . . . 102
9.3 Development story for D5: “The Inspection Valley” . . . 104
9.4 Development story for D3: “Implement First, Verify Later” . . . 105
9.5 Development story for D3: “Home Sweet Home” . . . 106
9.6 Development story for D5: “Curing the Window Plague” . . . 107

10.1 Visualizing working sets: visualization principles . . . 112
10.2 The principles of the force-based layout . . . 114
10.3 Example of the “U Can’t Touch This” pattern . . . 119
10.4 Two examples of the “Past: To Edit or Not To Edit” pattern . . . 120
10.5 Example of the “The Guiding Star” pattern . . . 121
10.6 Example of the “Stay Focused, Stay Foolish!” pattern . . . 122
10.7 Two examples of the “Moving in Circles” pattern . . . 123
10.8 Example of the “The Past Awakens” pattern . . . 124
10.9 Example of the “Multi-Part Session” pattern . . . 125
10.10 Example of the “Thirst for Knowledge” pattern . . . 126
10.11 Example of the “The Working Funnel” pattern . . . 127
10.12 Another example of the “The Working Funnel” pattern . . . 128

11.1 An example of the “Activity Forest” view . . . 130
11.2 An example of the “Activity Timeline” view . . . 131
11.3 An example of the “Cumulative Activity” view . . . 131
11.4 An example of the “Workspace View” . . . 132
11.5 Subsequent moments visualized through the “Workspace View” . . . 132
11.6 Cumulative Activity View for a bug-fixing session of Alice . . . 133
11.7 Part of the Activity Forest for a bug-fixing session of Alice . . . 133
11.8 Combined UI View and Activity Timeline for a bug-fixing session of Alice . . . 134
11.9 Two Workspace Views of the bug-fixing session of Alice . . . 134
11.10 Cumulative Activity View of the enhancement session of Bob . . . 135
11.11 Part of the Activity Forest of the enhancement session of Bob . . . 136
11.12 Three Workspace Views of the enhancement session of Bob . . . 136
11.13 The UI View of the enhancement session of Bob . . . 136
11.14 DFloWeb composed of (1) a Navigation Bar, (2) a Session Log, (3) a Timeline, and (4) the Visualization Canvas . . . 137
11.15 Visualization principles of DFloWeb . . . 138
11.16 Three time steps of the same session visualized with DFloWeb . . . 139

11.17 A Fraction of an Enhancement Session Depicted with DFloWeb 140
11.18 The Same Session of Figure 11.17 at a Later Time 140
11.19 DFloWeb Depicting a Bug-Fixing Session 141

12.1 A screenshot of *Pharo* IDE manifesting the Window Plague (top left) and the same environment after enabling the Plague Doctor (bottom right) 146
12.2 The Settings of the Plague Doctor 148

13.1 Main UIs to display code: (a) Browsers, (b) Debuggers, and (c) Message Lists 154
13.2 Visualizing a snapshot of a session (left) and the corresponding screen regions used to measure the chaos (right) 157
13.3 Session snapshots explained 158
13.4 Elision and Layout Strategies in a nutshell 161
13.5 Elision strategy for (a) Code Browsers, (b) Debuggers, and (c) Message Lists 162

14.1 Potential architecture for the infrastructure of Eye 170
14.2 Treemap of IDE interactions 173
14.3 The UI of the *Smalltalk-80* IDE (1983) compared with the *Pharo* IDE (2017) 174
14.4 The UI of *Eclipse* 1.0 (2001) compared with *Eclipse* Oxygen (2017) 174

A.1 The Blended City – Visualization principles and proportions 204
A.2 The same view of Figure A.1 with different ingredients (0% - 100% - 50%) 205
A.3 Color wheel and triadic color scheme 206
A.4 Linear color blend on triadic color scheme 207
A.5 Aging process: example in the Timeline 207
A.6 The architecture of the Blended City 208
A.7 View of the City with all the Activities 209
A.8 *Spec* and *Morphic* Market Districts 211
A.9 Changes in the *Pharo* system 212
A.10 The changes of GT-Tools Packages 213
A.11 A view of the system highlighting only stack traces and developer interactions 214

B.1 Session Digest: How have you spent your time? What did you do? 219
B.2 Activities and time components in a sunburst visualization 220
B.3 Activity View: Decomposing an high-level development activity 220

Tables

5.1 List of interaction data events recorded by DFLOW . . . 48

6.1 Dataset – Smalltalk sessions data per type . . . 55
6.2 Dataset – Smalltalk sessions data per developer . . . 56
6.3 Dataset – Java sessions data per developer . . . 57
6.4 Results – Amount of Understanding in *Smalltalk* sessions varying P_E and P_I . . . 63
6.5 Results – Amount of Understanding in Java sessions varying the estimate of P_E . . . 64
6.6 Results – Development activities per session type . . . 65
6.7 Results – Smalltalk development activities per developer . . . 65
6.8 Results – Java development activities per developer . . . 66

7.1 Dataset – Total values grouped by developer . . . 71
7.2 Dataset – Average values grouped by developer . . . 71
7.3 Dataset – Demographics of developers . . . 72
7.4 Results – Time components aggregated per developer . . . 78
7.5 Results – Correlation of Understanding time (UND) with the number of OI events (NOI) and the Duration of the time spent Outside the IDE (DOI) . . . 80

8.1 Dataset – Study on the Navigation Efficiency . . . 86
8.2 Results – Preliminary estimates for Navigation Efficiency . . . 88
8.3 Results – Navigation Efficiency with Δ-cost . . . 90
8.4 Results – Navigation Efficiency with UI-Aware navigation cost . . . 94
8.5 Results – An upper bound for Navigation Efficiency: UI-Aware Navigation Efficiency considering all the entities involved in a development session . . . 95

9.1 Dataset – Sessions statistics grouped by developer . . . 102
9.2 Dataset – Development events grouped by developer . . . 103
9.3 Dataset – Window information grouped by developer . . . 103
9.4 Results – Track and Flow characterization of developer session . . . 109

10.1 Dataset – Totals values and values aggregated per session . . . 117

13.1 Dataset – DFLOW dataset to characterize and measure chaos . . . 155
13.2 Space Occupancy Metrics . . . 157
13.3 Results – Distribution of Space Occupancy Metrics across all sessions . . . 158
13.4 Chaos-Levels: Comfy, Ok, Mess, and Hell . . . 159
13.5 Results – Average time spent per chaos-level . . . 159
13.6 Results – Chaos, UI, and Understanding Time . . . 160
13.7 Results – Percentage gain of Space Occupancy Metrics . . . 163
13.8 Results – Percentage gain and delta time . . . 164
13.9 Results – Average Weighted Overlapping per chaos-level . . . 164

A.1 Dataset – Source code changes in the considered period . . . 202

A.2 Dataset – Stack traces data in the considered period 203
A.3 Dataset – IDE interaction data in the considered period 204

Part I

Prologue

1

Introduction

Once upon a time people used punch cards to input data and sequences of instructions in programmable machines. The first programmable machine, the *"Jacquard loom"*, was invented in 1801 [McC99]. This revolutionary loom "reads" patterns from a punch card and automatically weaves them into the silk. Physical motion was replaced with electrical signals with the advent of the first electronic general-purpose computer: the *ENIAC* [HHG46]. Starting from the 1950s several programming languages were created. Notable examples include *assembly* that first appeared in 1949, *FORTRAN* (1957), *LISP* and *ALGOL* (1957), and *COBOL* (1959). Early computer programs, similar to the instructions of the Jacquard loom, were stored on punched cards. At this time *programming* was achieved by *physical motion*. When there was an error, for example, programmers used *physical patches* to correct wrong holes in punched cards by covering them. With the advent of modern programming languages, developers replaced punch cards with source code files. Programming became more and more an activity related to text files and compilers (or interpreters, depending on the language). After the *Compiler Era*, the next big revolution happens with the development of Integrated Development Environments, also known as IDEs.

An IDE is *"a large collection of integrated tools, each accessed through a uniform user interface"* [Seb12]. The early IDEs are *Dartmouth BASIC* (1964) with its interactive command line, *Smalltalk*-80 (1980) with the first graphical user interface (GUI), and *Turbo Pascal* (1983). Since then, a plethora of different IDEs has been developed. Some IDEs are very specific for a language, while others provide efficient multi-language support. The birth of the Java programming language, for example, originated a fight between *Eclipse*[1] and *IntelliJ IDEA*[2] to determine the dominant Java IDE [Gee05]. Eclipse, originally developed by IBM and handed over to the Eclipse Foundation in 2001, turned out to be the most appreciated Java IDE [Gee05]. One of the main reasons for success was its plug-in architecture that enables developers to develop tools to extend the current IDE capabilities [Gee05]. However, developers have plenty of other IDEs to choose from. According to the *Top IDE Index* [Car16], the most used IDEs are *Visual Studio*, *Eclipse*, *Android Studio*, *Vim*, and *NetBeans*. The 5 most used languages in 2017 are *Java*, *C*, *C++*, *C#*, and *Python* [TIO17].

Today, most developers use their favorite IDE to manipulate source code and to perform the vast majority of their software development activities [GLD05, GGD07, Seb12]. *Codeanywhere, Inc.*,[3] for example, collected answers from two thousand developers and reported that the most

[1] See `https://www.eclipse.org`
[2] See `https://www.jetbrains.com/idea`
[3] See `http://codeanywhere.com`

popular tools used by developers are *Notepad++*, *Sublime Text*, *Eclipse*, *NetBeans*, *IntelliJ IDEA*, and *Vim* [COD15]. Among them, there are IDEs and highly configurable stand-alone text editors. *Text editor aficionados*, in fact, use tools such as *Vim* or *Emacs*, instead of IDEs. Our conjecture is that these developers reached such a high level of efficiency in manipulating source code with text editors that they do not need the tools and facilities offered by IDEs. Even though it would be interesting to investigate why some developers prefer text editors to IDEs, we only focus on developers using IDEs to support their development activities.

IDEs aim at easing the development and maintenance of software systems by providing different tools and facilities to support various kinds of activities [KMCA06, SMDV08]. Developers use IDEs to read, understand, and write source code. For reading and writing, the most used tool is the *code editor* (or *code browser*). To understand code—in addition to reading it—developers also take advantage of tools such as the debugger, the package explorer, or the reference browser. It has been showed that developers spend more time reading code than writing it [VMV95]. Reading code is the building block of program comprehension which has been estimated to occupy more than half of the working time of a developer, *e.g.,* [ZSG79, FH83, Cor89, MML15b]. In addition to being time consuming, program comprehension is also one of the most challenging tasks performed by developers [LVD06]. To understand source code, developers need to navigate the software space [KM05]. In this process, they construct a *mental model* of the system [SLVA97, Wal03, RCM04, KMCA06], *i.e., a link between the source code and their mental representation* [FGS11], which is essential to support source code comprehension.

Navigating, reading, understanding, and writing source code inside the IDE are high-level activities which are composed of several low-level events, known as *"IDE interaction data"* [KM05, MKF06]. Examples include opening a code editor on a method, inspecting an object while debugging, moving the cursor of the mouse, or writing a new line of code in the body of a method. IDE interactions capture the intentions of developers and manifest their mental models [GSBS14]. For this reason, we believe that leveraging this information inside the IDE can provide benefits to different phases of the development process. In addition, this data can be analyzed retrospectively to better understand the behavior of developers inside the IDE. Existing research already showed the importance of interaction data. Frey *et al.*, for example, claimed that future program investigation tools need to track the way developers navigate code to support software engineering activities [FGS11]. According to Murphy *et al.*, interaction data can be used to evolve IDEs according to user needs [MKF06]. Following this intuition, IDEs need to be *aware* of interaction data and intensively exploit it to support the development workflow.

However, creating an *"Interaction-Aware IDE"* is a very demanding goal. In the first place, since IDEs neglect this information, the first step of our research consists in modeling and persisting interaction data. For this purpose we developed DFLOW: A framework to model, profile, and persist IDE interactions [ML13a, Min14, MML15b]. Interaction data is a largely unexplored source of information. In its raw form, it is a long stream of events and developers daily generate thousands of such events. The second challenge of our research is therefore to interpret these streams of events. To this aim we devised various models to reconstruct, for example, high-level programming activities from interaction histories or measure how efficient are developers in navigating the software space. The last, and probably the most challenging, part of our research consists in supporting software development with interaction data. Examples include visualizations to ease the navigation of previously interacted program entities and means to adapt the user interface of the IDE.

1.1 Our Thesis

We formulate our thesis as follows:

> *"Interaction-Aware Development Environments enable novel and in-depth analyses of the behavior of software developers and set the ground to provide developers with effective and actionable support for their activities inside the IDE."*
>
> Roberto Minelli, 2017

To validate our thesis we implemented DFlow, a framework to model and record the interaction data events happening inside the IDE [ML13a, Min14, MML15b]. On top of DFlow we devised various approaches to interpret and leverage this novel source of information.

1.2 Contributions

The contributions of our research can be grouped in two high-level categories: i) modeling & analyzing interaction histories and ii) supporting tools.

1.2.1 Modeling & Analyzing Interaction Histories

- We devised models to reconstruct development activities from interaction data to understand how developers spend their time inside the IDE [MMLK14, MML15b];
- We devised approaches to model source code navigation efficiency in the IDE and applied them on a large dataset of development sessions [MML16a];
- We devised visual approaches to understand different aspects of developers interactions, such as the workflow of developers and how they use the GUI of the IDE [ML13b, MMLB14, MBML14];
- We devised an approach to visualize the evolution of the working set, *i.e.,* the program entities involved in a development session [MML16b];
- We studied how the entropy of the user interface of an IDE evolves and we proposed a mechanism to tame it [MMRL16];
- We defined the concept of *"Self-Adaptive IDEs"* and envisioned how they can leverage interactions with different information sources to support the workflow of developers [Min14];
- We devised an approach to visualize three data sources at once: interaction data, fine-grained source code changes, and stack traces [SMML15];
- We envisioned how to use interaction data to introduce a gamification layer on top of the IDE [MML15a].

1.2.2 Supporting Tools

- We developed DFlow, a non-intrusive profiler that records IDE interactions and makes them available for further use [ML13a, Min14, MML15b];
- On top of DFlow, we developed The Plague Doctor [MML15c], a tool that leverages interaction data to reduce the so called *window plague* [RND09].

1.3 Outline

This section presents the outline of our dissertation, which is structured in 5 parts.

Part I: Prologue. The first part of our dissertation sets the ground for our research by introducing interaction data and the related fields.

Chapter 2 (p. 9) introduces interaction data, the main source of information of our research, and traces the history of programming, from punch cards to IDEs.

Chapter 3 (p. 17) details the *Pharo* IDE, the target IDE for our research. A reader familiar with this IDE can safely avoid reading this chapter. The chapter details the object model of *Pharo*, its UIs, and explains why we chose *Pharo*.

Chapter 4 (p. 27) presents an overview of the research fields interested by this dissertation. This includes existing approaches to record and to analyze interaction data, fine-grained source code changes, biometric data, and approaches to support developers inside the IDE. Finally, the chapter addresses ethical issues arising from the collection of development data.

Part II: Modeling, Recording, and Interpreting Interaction Data (p. 45). This part describes how to record and interpret interaction data, which is the key to make it available for further use.

Chapter 5 (p. 45) discusses the contributions of our research in recording interaction data inside the IDE. In particular the chapter presents DFlow, the IDE interaction profiler we developed to support our research.

Chapter 6 (p. 53) describes how we can use interaction histories to estimate how developers spend their time. In particular, the chapter focuses on the case of program comprehension, one of the core activities of software development.

Chapter 7 (p. 69) refines the model introduced in Chapter 6 and explains how to use interaction histories to reconstruct high-level development activities.

Chapter 8 (p. 83) focuses on the use of interaction data to model and measure how efficiently developers navigate source code.

Part III: Visual Analytics of Development Sessions (p. 99). The third part of our dissertation describes a number of visual approaches to gather further insights from interaction histories.

Chapter 9 (p. 99) illustrates a visual approach to better understand how developers use the user interface of the *Pharo* IDE.

Chapter 10 (p. 111) presents a visual approach to understand the evolution of working sets during development sessions.

Chapter 11 (p. 129) illustrates a catalog of visualizations to support visual storytelling of development sessions.

Part IV: Supporting Developers with Interaction Data (p. 145). The fourth part of our dissertation describes how to leverage interaction data to support software development inside the IDE.

Chapter 12 (p. 145) presents the PLAGUE DOCTOR: A tool built on top of DFLOW that support developers by mitigating the so-called *window plague* [RND09].

Chapter 13 (p. 151) explains how to use interaction data to characterize and quantify the "level of chaos" in an IDE and proposes an approach to tame it.

Part V: Epilogue (p. 169). The fifth part concludes our dissertation by summarizing the work and highlighting future directions for our research.

Chapter 14 (p. 169) outlines possible research directions for the future.

Chapter 15 (p. 177) summarizes and concludes our work.

Appendices (p. 201). At the end of this dissertation we include two side works we carried on during our research.

Appendix A (p. 201) details a visual approach that depicts multiple concerns concurrently by blending them together. Our approach considers interaction data, source code changes, and stack traces.

Appendix B (p. 217) presents a vision that uses interaction data to introduce a gamification layer on top of the IDE: the "*Development Empire*".

2

IDEs and Interaction Data

This chapter introduces the context of our research: Integrated Development Environments and *interaction data*, *i.e.,* all the events that programmers carry out during development. Nowadays most developers use an IDE as their main vehicle to develop software systems. At the beginning of this chapter we take a step back and trace the history of programming, explaining how we moved from punch cards to modern development environments.

During development, programmers interact with a number of facilities and tools: web browsers, bug trackers, IDEs, mail clients, *etc.* In practice, all these activities generate thousands of events that we can group under the name of *interaction data*. In this chapter we briefly describe why and how we believe in the importance of this largely unexplored source of information to both retrospectively analyze the behavior of developers and provide them with effective and actionable support inside and outside the IDE. Our research focuses on the interactions happening inside the IDE. For this reason, this chapter emphasizes and details this specific type of interactions.

Structure of the Chapter

Section 2.1 traces the history of programming, from programmable looms to IDEs. In Section 2.2 we introduce interaction data and refine the scope to the interactions happening inside the IDE, the real core of our research. Finally, Section 2.3 summarizes and concludes the chapter.

2.1 From Punch Cards to Modern Development Environments

Once upon a time people used punch cards (or perforated cards) to input data and sequences of instructions in programmable machines. In 1801, Joseph-Marie Jacquard devised a programmable loom, known as *"Jacquard loom"*, to automatically weave silk by reading patterns from a series of punch cards [McC99].

> *"We use the term 'standing on the shoulders of giants.' They aren't making components but they use existing components. So, if anything, [IDEs] made great programmers greater, but also made good programmers better."*
>
> — David "I" Intersimone
> Delphi Evangelist

Inspired by the work of Joseph-Marie Jacquard, in 1837 Charles Babbage designed the *"Analytical Engine"* considered the first general-purpose programmable computing engine [Swa16]. According to its design, developers would use punch cards to input data in the machine, the same mechanism of the Jacquard loom. Babbage never managed to complete the construction of this machine at his time. A few years later, Ada Lovelace specified a method for calculating Bernoulli numbers with the Analytical Engine. This is often recognized as the world's first computer program [FF03].

The first form of *programming*, thus, was performed through physical motion: punched cards, knobs, and switches. In 1942 physical motion was replaced by the electrical signals when the US Government built the so-called ENIAC: the first electronic general-purpose computer [HHG46]. In the same period John von Neumann developed two concepts that shaped the development of programming languages: shared-program technique and conditional control transfer [CCH17]. The former states that hardware should be simple and not tailored to every single program. Instead, the complexity should be in the instructions to control the hardware. Conditional branching, or conditional control transfer, states that instructions should not be necessarily executed in sequential order, *i.e.,* the execution of a program can be altered by logical branches (*e.g., If-then* expressions). John von Neumann is also known for the *stored-program digital computer* (or *Von Neumann Architecture*) [CCH17]. In this model, program instructions and data are stored in the same memory location enabling instructions and data to be modified in the same way. The first machine implementing this concept was the EDVAC (Electronic Discrete Variable Automatic Computer), developed by John Mauchly and Presper Eckert, who previously designed the ENIAC [vN93].

Programming with *assembly* languages was very tedious and error prone. For this reason, the 1950s see the birth of more practical alternatives to assemblers: the so-called high-level programming languages. *Short Code*, developed by John Mauchly in 1949, was the first such language [Seb12]. Unlike assembler, in which there is a strong correlation between the language and machine code instructions, in *Short Code* each statement represents a mathematical expression in human-readable form. This language, however, has to be translated to machine code and, as a result, was significantly slower than machine code. In the same period, Alick Glennie developed *Autocode*, often considered as the first language to use a compiler[1] to automatically convert the

[1] The answer to "*Who wrote the first compiler?*" is very controversial: In 1952 Alick Glennie developed Autocode with its compiler and Grace Hopper finished her compiler for the A-0 System.

language into machine code [Ben12]. In 1957 it was the turn of *FORTRAN*, a high-level general purpose imperative programming language designed and developed by John Backus at *IBM* [Bac78]. Thanks to its performance for computationally intensive applications, *FORTRAN* is still used today in cutting-edge research [Phi14]. Examples include atmospheric modeling and weather prediction carried out by the National Center for Atmospheric Research (NCAR), classified nuclear weapons and laser fusion codes at Los Alamos & Lawrence Livermore National Labs, and NASA models of global climate change [Phi14].

A few years later, a team led by Grace Hopper developed *FLOW-MATIC*, recognized as the first programming language to use English-like statements to express operations [Sam69]. In the same period John McCarthy (MIT) developed *LISP* and a consortium called *CODASYL*[2] developed *COBOL*. As the expansion of the acronym suggests (*i.e.,* "LISt Processor"), *LISP* was invented to efficiently manipulate lists, *COBOL*, instead, was intended for business use. *LISP* pioneered many of the concepts that are now considered the foundations of software engineering, such as tree data structures, automatic storage management, dynamic typing, higher-order functions, and recursion [Dal17]. In the following years were defined most of the fundamental programming paradigms that are still in use today. *ML* was the father of all the statically-typed functional programming language, *Prolog* was the first logic programming language, and *Smalltalk* sets the ground for object-oriented languages [Kay93]. *Smalltalk*'s development started in 1969 by Alan Kay, Dan Ingalls and Adele Goldberg at Xerox Palo Alto Research Center (PARC). The first public version, *Smalltalk*-80, appeared only after ten years. A peculiarity of *Smalltalk* is that it provides a *live* development environment featuring powerful debugging and inspection tools.

> *"A programming environment is the collection of tools used in the development of software. This collection may consist of only a file system, a text editor, a linker, and a compiler. Or it may include a large collection of integrated tools, each accessed through a uniform user interface.*
>
> *In the latter case, the development and maintenance of software is greatly enhanced."*
>
> — Robert W. Sebesta [Seb12]

A "programming environment" is a set of tools to support software development. For example, the *UNIX* environment released in the 1970s, includes tools to create, run, and maintain software. Essential tools are a text editor and a compiler (or interpreter, depending on the language). The *"large collection of integrated tools, each accessed through a uniform user interface"* is better known as *"Integrated Development Environment"* or IDE. Together with *Smalltalk*, also other languages started to provide similar environments. In a broader sense, the very first language that implemented an IDE was *Dartmouth BASIC* back in 1964. It provided an interactive command line interface (CLI) that integrates editing, compilation, debugging, execution, and file management comparable to modern IDEs. The advent of the Graphical User Interface (GUI), however, changed everything. The *Alto* personal computer, developed at Xerox PARC in 1973, was the first computer to implement the desktop metaphor and to provide a GUI controlled with a mouse. A few years after, Steve Jobs and Jef Raskin evolved these ideas to realize their *Apple*

[2] *CODASYL* stands for "Conference on Data Systems Languages". It was an organization founded in 1957 by the U.S. Department of Defense aimed at developing programming languages.

Lisa, in 1983. In the same year, Borland Ltd. launched *Turbo Pascal*: An IDE for *Pascal* that lets developers write, compile, and debug code. In 1985 Microsoft launched the first version of its Windows operating system. However, we have to wait until the early 90s to have a stable and usable version (Windows 3.0 and 3.1). In the same period, Microsoft released *Visual Basic* (VB), sometimes wrongly credited as the first IDE.

2.1.1 From the 1980s to the Present Day

The 1980s and 1990s see the rise of IDEs. With its Graphical User Interface (GUI), powerful debugger and inspection tools, *Smalltalk-80* is often recognized as the first IDE. Other two examples of early IDEs were *Turbo Pascal* developed in 1983 and *Visual Basic* in 1991.

> *"If Visual Basic hadn't happened,*
> *Delphi wouldn't have happened,*
> *and Visual Studio wouldn't have happened,*
> *and PC guys would be pounding out forms*
> *textually."*
>
> — Jeff Duntemann [Pat13]
> Former Employee at Xerox Corporation

In 1995 Borland—that previously developed *Turbo Pascal*—released *Delphi*, an IDE aimed at building applications rapidly by visually dragging and dropping components. In 1997 Microsoft released its new IDE called *Visual Studio*. This IDE evolved until the present day and today is one of the most used IDEs worldwide. *Eclipse* and *JetBrains IntelliJ IDEA*, both released in the early 2000s, evolved until the present day, and are among today's most used IDEs.

According to the *Top IDE index* [Car16], a ranking created by analyzing data coming from Google Trends,[3] today the five most popular IDEs worldwide are *Microsoft Visual Studio*,[4] *Eclipse*, *Android Studio*,[5] *Vim*,[6] and *NetBeans*.[7] *Codeanywhere, Inc.*,[8] surveyed more than 10,000 developers to discover the most popular tools and IDEs in practice [COD15]. *Notepad++*,[9] *Sublime Text*,[10] *Eclipse*, *NetBeans*, and *IntelliJ IDEA* are the five most used *tools* employed to develop source code. It is interesting to notice that in both studies emerge the fact that some developers prefer to use a highly configurable text editor, such as *Vim* or *Sublime Text*, instead of a full fledged IDE to develop source code. Even though this would be an interesting phenomena to investigate, in our dissertation we only focus on developers using IDEs to support their development activities.

2.1.2 Summing Up

In the last 200 years programmable looms and punch-cards evolved into full-fledged integrated development environments. Nowadays developers have all the tools and facilities to develop

[3] See `https://www.google.com/trends`
[4] See `https://www.visualstudio.com`
[5] See `https://developer.android.com/studio`
[6] See `http://www.vim.org`
[7] See `https://netbeans.org`
[8] See `http://codeanywhere.com`
[9] See `https://notepad-plus-plus.org`
[10] See `https://www.sublimetext.com`

software systems at their fingertips. While developing software system, developers perform different kinds of high-level activities in the IDE, such as navigating, reading, and understanding code. These activities are composed of several low-level events, known as *"IDE interaction data"* [KM05, MKF06], that are the essence of our research.

The following section provides more details on interaction data and explains why we believe in the importance of this largely unexplored source of information.

2.2 What is Interaction Data? Why is it Important?

We call "interaction data" the information *encapsulated* in an *interaction* between two subjects. In our context, one of the two subjects is represented by the developer sitting in front of her workstation carrying on her development activities. The other subject, however, can be of heterogeneous nature. In a workday, developers interact with various tools and facilities, such as IDEs, web browsers, mail clients, bug trackers, *etc.* as depicted in Figure 2.1.

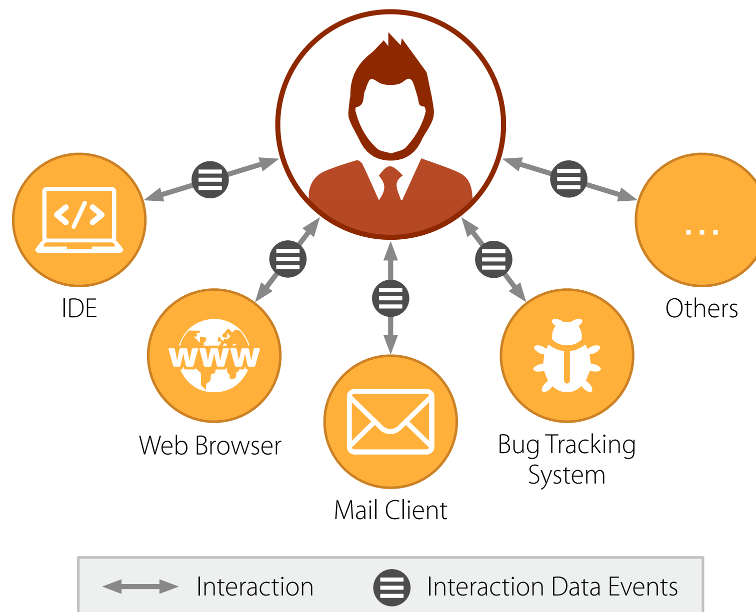


Figure 2.1. Interactions between the developer and different sources

Interacting with these sources generates thousands of events, that we call *interaction data events*. If we consider the web browser, for example, users perform interactions such as opening a new tab, typing a website in the address bar, adding a website to bookmarks, *etc.* In essence, these events precisely capture the actual behavior of developers (or users). By leveraging this source of information we can better understand the habits of developers and potentially supporting them by finding, and isolating, the bottlenecks in their ordinary workflow.

Unfortunately, interaction data is hardly available and largely unexplored. For example, IDEs do not record the interactions of developers while carrying out their development activities. To overcome this limitation, researchers developed tools and plug-ins to partially keep track of interaction data, such as MYLYN [KM05]. The interactions between users and tools are of ephemeral nature, making it hard to directly exploit their potential. Thus, the first challenge for our research is to persist these interactions, as discussed in Chapter 5.

Considering the interactions with all the available sources of information makes interaction data an extremely broad domain. To restrict the field, we concentrate our research only on the interactions happening inside the IDE, or IDE interaction data,[11] as detailed in the next section.

2.2.1 Interactions with the IDE

IDEs ease the development and maintenance of software systems by offering tools and facilities to support different development activities [KMCA06, SMDV08]. Developers use the IDE to navigate, read, understand, and ultimately write source code. To do so, they have to *interact* with various user interfaces such as the *code editor* (or *code browser*), the *debugger*, and the *package browser*. For example, when a developer modifies the source code of a method, she will press a sequence of keystrokes to change the body of the method and she may also re-arrange the user interfaces of the IDE to better support code editing activities. We distinguish different *types* of events in a taxonomy, detailed in Chapter 5, that includes *meta* and *low-level* events. Figure 2.2 depicts the flow of interactions between the developer and the IDE.

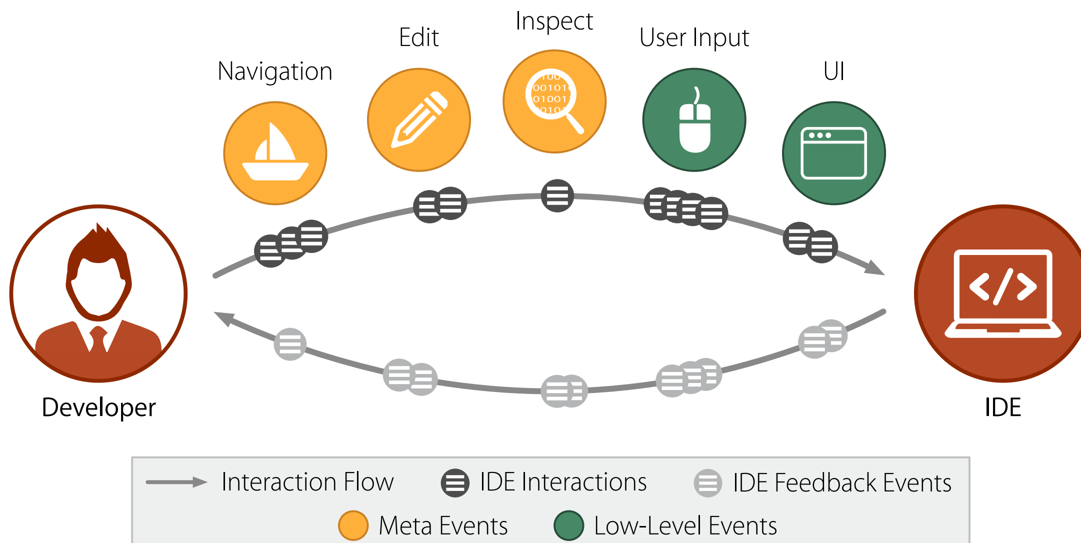


Figure 2.2. The flow of interactions between the developer and the IDE

A development session is made of a continuous *feedback-loop* between the developer and the IDE: The developer "asks" something to the IDE and receives an "answer" from it. All the requests that a developer forwards to the IDE are, in essence, interaction data. We call the answers supplied by the IDE *"feedback events"*. For example consider a scenario where Alice wants to edit the source code of a method. In the first place she has to reach the method of interest. To do so, she can either jump directly to the method (*i.e.,* by using the search capabilities of the IDE) or perform a sequence of *navigations*, likely following structural relationships between code entities (*i.e.,* in *Eclipse*, for example, she can use the "Package Explorer"). In both cases, these two high-level activities, are composed of several low-level events, that are known as *"IDE interaction data"* [KM05, MKF06]. For example, to trigger the search interface, she can either press a keyboard shortcut (*e.g.,* ⌘ +'F') or click on an entry in a contextual menu in the IDE window (*e.g.,* Edit ▹ Find). In both cases, Alice will perform sequences of mouse movements,

[11] For the sake of readability, from now we may omit the term "IDE" and use the general expression *"interaction data"* to indicate all the interactions happening inside the IDE.

clicks, and keystrokes (or keyboard shortcuts). In turn, the IDE will provide her with *answers*, for example, by popping out the search user interface when requested. Then she types in the query, by means of a series of keystrokes, and the IDE answers by opening a user interface with the results. To complete her tasks, Alice will continue to interact with the IDE generating, by the end of the day, thousands of interaction events. These events are the core of our research.

2.3 Programmable looms, IDEs, and Interaction Data

This chapter summarized the history of programming, from the *Jacquard loom* to the first general-purpose programmable computing engine, until the raise of modern IDEs.

Nowadays most developers use an IDE to develop software systems. While doing so they generate millions of interaction events that precisely capture their behavior. In this chapter we introduced this largely unexplored source of information and explained why we believe it is important. The aim of our research is to study and leverage the flow of interactions between developers and IDEs. However, as discussed in the chapter, developers have many different IDEs to choose from. In our research, we decided to target the *Pharo* IDE, an open-source IDE that supports a language inspired by *Smalltalk*. The next chapter details this IDE and explain how it differs from mainstream IDEs.

3

The Pharo IDE

THIS CHAPTER details *Pharo*, the target environment for our research, which is both an object-oriented programming language inspired by *Smalltalk* and an open-source IDE. Throughout the chapter we discuss what is an "*image*", explain the rules underlying the *Pharo* object model, and introduce some terminology. Later we detail the various user interfaces offered by this IDE to support development activities. Examples include the *code browser*, the *playground*, and the *debugger*. We conclude the chapter by motivating why we carried out our research in *Pharo*.

This chapter presents the *Pharo* IDE to ease the comprehension of the rest of this dissertation. A reader familiar with this IDE can safely skip this chapter.

Structure of the Chapter

Section 3.1 introduces the *Pharo* IDE and its image-based architecture. Section 3.2 details its object model and source code organization. In Section 3.3 we discuss the most used user interfaces of the *Pharo* IDE while in Section 3.4 we explain why we decided to work in *Pharo*.

3.1 What is *Pharo*?

Pharo is both a programming language and a development environment.[1] The *Pharo* language is inspired by *Smalltalk*—the father of all object-oriented programming languages [Kay93]. Its user interface is a window-based environment focused on simplicity and immediate feedback. The example depicted in Figure 3.1 shows the *Pharo Main Window* with 6 different user interfaces (*i.e.*, windows) opened. In a development session, on average, developers interact with 24 windows [MMRL16]. At any given moment, only one of these windows is in focus and acts as target for all user interactions (Window D in Figure 3.1).

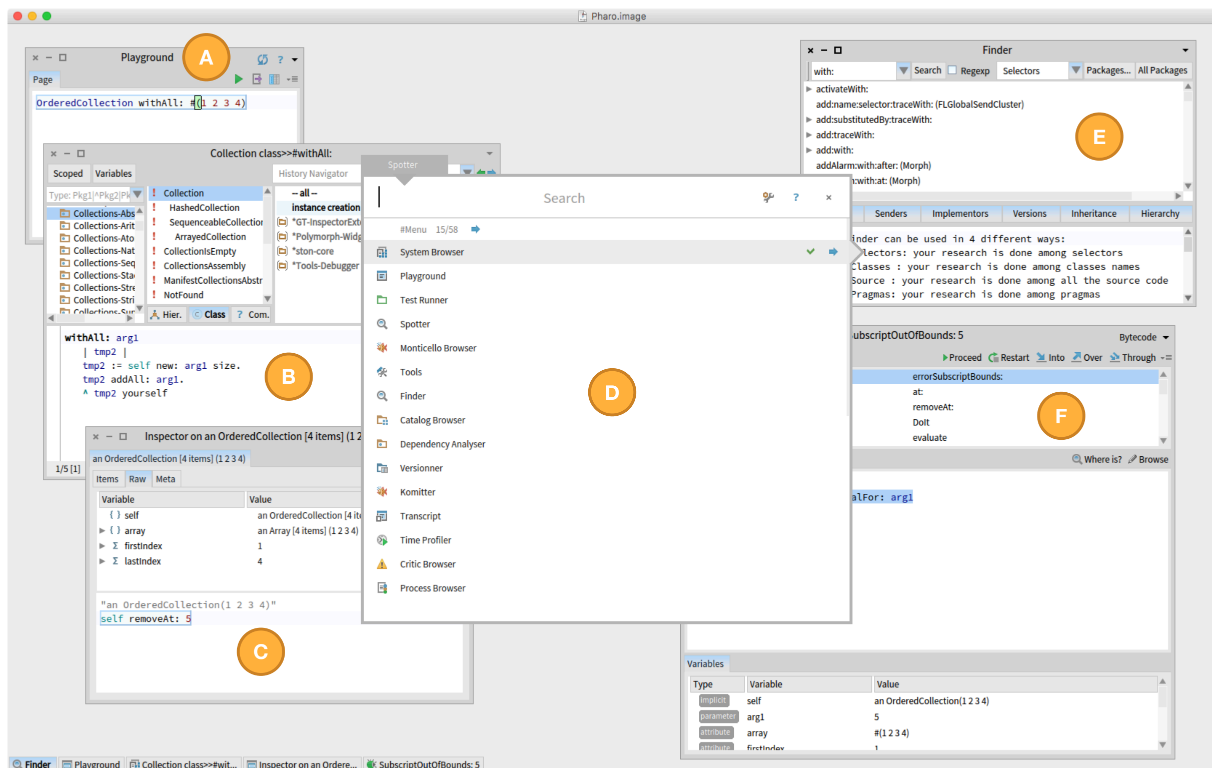


Figure 3.1. The Pharo Main Window and its UIs: (A) Playground, (B) Code Browser (or Code Editor), (C) Inspector, (D) *Spotter* Search Interface, (E) Finder UI, and (F) Debugger

Pharo is *Image-Based*: It combines code and data in a single cross-platform file known as *"Image"*. An Image is a snapshot of an entire running system at any given point in time. It contains the state of all the objects of the system at that moment, including classes and methods, since they are also objects. Every time a developer closes the Image, *Pharo* freezes the state of all the objects. When the developer opens the *Pharo* image again, the system restores all the objects with their state.

Besides the Image file, there is another file that composes the *Pharo* environment: the *Changes* file. This file logs all the source code changes happening in the system.

[1]See `http://pharo.org`

3.2 The Object Model of Pharo

Pharo is an object-oriented language and, as such, to combine state and behavior, developers use high-level constructs called *objects*. The *Pharo* language is inspired by *Smalltalk*. Its object model relies on 10 simple rules [DZHC17]. For example, Rules 1 and 2 say that "*everything is an object*" and that "*every object is an instance of a class*". This captures the *"pure"* essence of object-oriented programming being lean, simple, elegant, and uniform [DZHC17]. Primitive values such as integers, booleans, and characters are also instances of their corresponding classes, *i.e.,* objects. *Classes* are no exception: They are also objects, thus instances of other classes: "*Every class is an instance of a metaclass*" (Rule 6). In *Pharo*, "*the metaclass hierarchy parallels the class hierarchy*" (Rule 7).

Except for a few syntactic elements, "*everything happens by sending messages*" (Rule 4). Messages are similar to methods in other object-oriented languages. A message is composed of i) a *selector* and ii) some (optional) message *arguments*. A message is sent to a *receiver*. The combination of a message and its receiver is called "*message send*". There are three types of messages: unary, binary, and keyword. *Unary messages* do not take parameters, *e.g.,* `Object new`. *Binary messages* always involve two objects, *e.g.,* `100 + 20`.[2] *Keyword messages* consist of one or more keywords, each ending with a colon and taking an argument, *e.g.,* `aPoint x: 10 y: 32`. Message sends can be chained: For example, to initialize a new `Person` called Roberto that is 29 years old one can invoke the constructor (*i.e.,* `new`) followed by the message `name:age:` as follows: `Person new name: 'Roberto' age: 29.`

3.2.1 Source Code Organization

Pharo organizes source code in *packages*, *classes*, *metaclasses*, *methods*, and *protocols*.

- A **package** is a group of related classes and methods.
- A **class** defines the structure (*i.e.,* variables) and the behavior (*i.e.,* methods) of its instances. Classes are objects and thus instances of a class (Rules 1 and 6).
  - In *Pharo*, each class is the unique instance of its **metaclass**. In the object model there is a metaclass hierarchy parallel to the standard class hierarchy. The developer has full access to the so called *"class side"*, thus she can and navigate and modify metaclasses and their methods. A common use of a metaclass is to create custom constructors, instead of using the ordinary `new` method to create a new instance of a class.
- A **protocol** (or **category**) is a group of methods sharing the same intent. It is mostly used for documentation purposes. A common established protocol is called *"accessing"* and groups all the accessors (*i.e.,* getters and setters) of a class.
- **Message**, **selector**, or **method** are terms used interchangeably to indicate an operation that can be performed by an object. A common notation to identify a message is `P1.Foo»#bar` which denotes the message `bar` of class `Foo`, contained in package `P1`.
- Finally, *Pharo* provides local and shared **variables**. Local variables belong to an object while shared variables can be shared globally, between a class, its subclasses, and its instances (*i.e.,* class variables), or between a group of classes (*i.e.,* pool variables).

[2] The `+` is a message, not an operator.

3.3 The Most Used UIs in the Pharo IDE

Pharo provides developers with different kinds of windows that support one or more development activities. This section lists the principal UIs of *Pharo* and explains their main functionalities.

3.3.1 Code Browser

The *Code Browser*, depicted in Figure 3.2, lets the user perform the most essential development activities: navigating, reading, and writing code [GGD07]. This user interface is composed of an upper and a lower part, each aimed at different purposes.

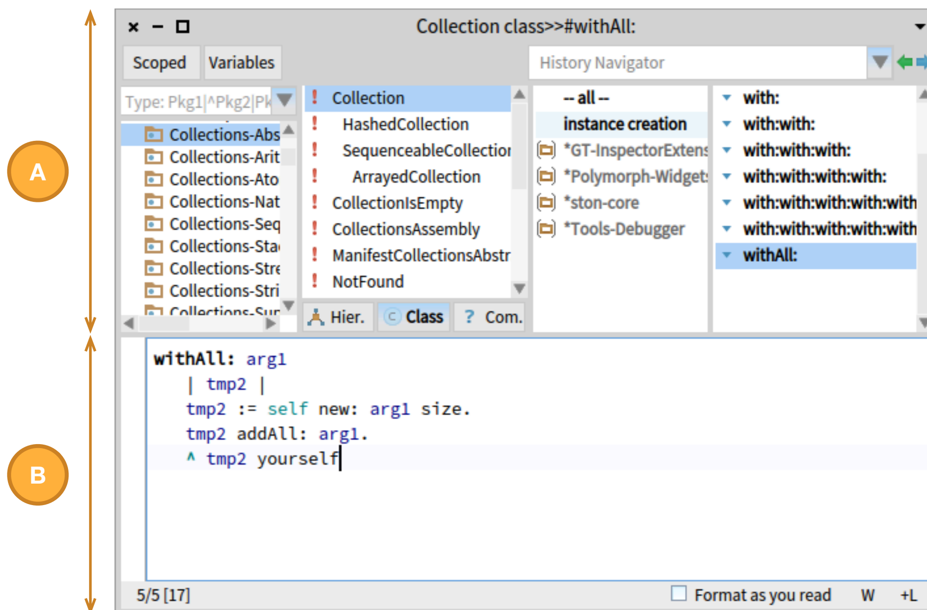


Figure 3.2. The Code Browser of the *Pharo* IDE

The upper half of the code browser (see Fig. 3.2-A) features four columns that let the user perform structural source code navigation. To navigate to the source code of a method a developer first selects the package that contains the class of interest in the first column. Then she selects the class in the second column, and finally the method on the last column of the browser. We detail the mechanics of navigation in Chapter 8, where we discuss how to measure the navigation efficiency of developers. The example in Figure 3.2 depicts a code browser after the navigation to the method `#withAll:` belonging to the `Collection` class, that is contained in the `Collections-Abstract` package. The third column lists the so-called *protocols* (or method categories). Its purpose is grouping related methods together. In the example, the method `#withAll:` has been categorized as *"instance creation"* because this method is responsible to instantiate a new `Collection` object. The selection of the protocol is optional.

Once the developer selects a method, the lower part of the code browser (see Fig. 3.2-B) displays its source code. The developer can use this text area to read and modify source code.

3.3.2 Workspace and Playground

Figure 3.3b depicts a *Workspace* (see Fig. 3.3b-A) and a *Playground* (see Fig. 3.3b-B). This UIs enable developers to run snippets of code. A common usage scenario for these UIs is to instantiate

a new object on-the-fly. In the example of Figure 3.3b, the developer uses a Workspace (and a Playground) to initialize a new `OrderedCollection` from the elements of an `Array`.[3]

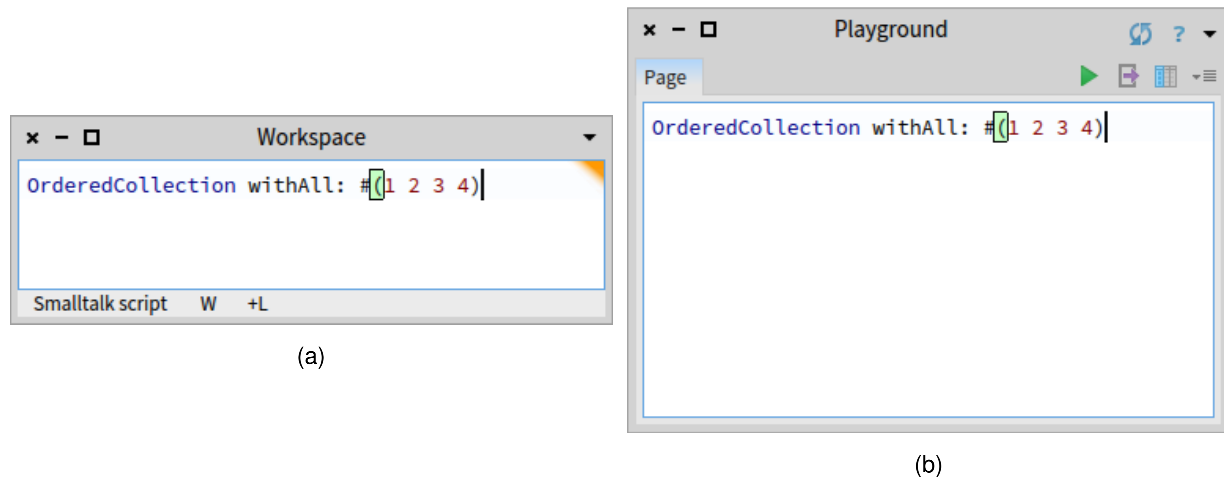


Figure 3.3. The Workspace (3.3a) and the Playground (3.3b) of the *Pharo* IDE

The Workspace has been almost completely replaced by the Playground in the *Pharo* IDE. The Playground, in fact, has the same capabilities of the Workspace and provides also additional features. For example, it *remembers* all the code snippets written by the developer and it enables developers to navigate them. However, when we carried on our research and data collection, developers had only the Workspace at their disposal. For this reason, the majority of our data only considers interactions with the Workspace and not with the Playground.

3.3.3 Inspector

Another key UI in the *Pharo* IDE is the *Inspector*, depicted in Figure 3.4. As the name suggests, this UI enables developers to *inspect* instances of objects. In the example below we are inspecting the object resulting from evaluating the code in Figure 3.3b that initializes an `OrderedCollection` with four `Integers`: 1, 2, 3, and 4.

In *Pharo* everything is *live*, thus there is no distinction between compile- and run-time. This enables the developer to open an Inspector on every object and interacting with it (*e.g.,* by adding elements to an array on-the-fly). The top part of this UI (see Fig. 3.4-A) lets developers browse the contents of the inspected object, similar to what other IDEs offer in their debug mode (or perspective). In this example, the inspected `OrderedCollection` contains 4 items (internally represented by an `Array` object and two variables: `firstIndex` and `lastIndex`).[4]

The lower part of the Inspector (see Fig. 3.4-B) can be used to query the inspected object. To do so the developer can write any snippet of code that will be executed on the currently inspected object, *i.e.,* `self` is bound to it. Queries can serve two purposes: Developers can either perform general inquiries to gather a better understanding of the object or execute code that modifies the current object on the fly. In the example in Figure 3.4, the developer wants to remove the element at index 5 from the currently inspected `OrderedCollection`.

[3] The *Smalltalk* expression `#(1 2 3 4)` initializes an `Array` containing four `Integers` 1, 2, 3, and 4.
[4] In *Smalltalk* the first element of a collection is at index 1 and not 0.

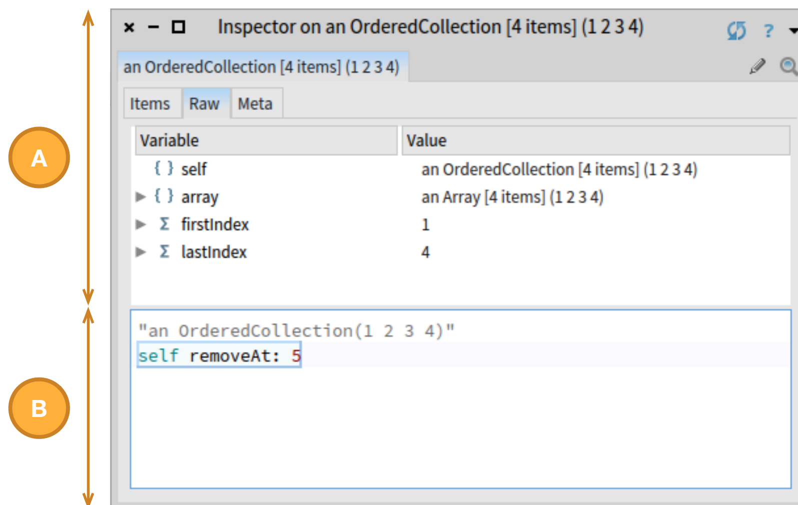


Figure 3.4. The Inspector of the *Pharo* IDE

3.3.4 Debugger

Buffer overflows are among the most common errors encountered by programmers. In Figure 3.4 the developer attempted to remove the fifth element from a collection of size 4. This triggers a `SubscriptOutOfBounds` error, displayed in a *Debugger* window, shown in Figure 3.5.

The upper part (see Fig. 3.5-A) lets the user browse the methods on the call stack, the middle part (see Fig. 3.5-B) displays the code of the method selected in the call stack and enables the user to modify it on-the-fly, and the lower part acts as a simplified inspector (see Fig. 3.5-C).

3.3.5 Search User Interfaces: Finder and Spotter

The *Finder*, depicted in Figure 3.6a, is one of the two UIs offered by developers to search for code artifacts. This UI lets developers find code entities by performing a search among selectors (*i.e.,* methods), classes, pragmas (*i.e.,* code annotations), or all the source code present in the *Pharo* image. The user can also enter regular expressions to perform more advanced queries. The *Finder* also enables developers to restrict the scope of the search to a subset of Packages.

Figure 3.6b shows *Spotter*, the other search UI of the *Pharo* IDE [SCG+15]. It is a unified search interface that combines different search tools into one. *Spotter* was introduced in a recent restyling of the *Pharo* IDE and aims at replacing the *Finder* UI. *Spotter* is a new tool and our dataset does not contain data about its usage. Kubelka *et al.* conducted a study to understand how developers use this tool [KBC+15].

3.3.6 Senders and Implementors Browsers

Figure 3.7 depicts the *Senders* (3.7a) and *Implementors Browsers* (3.7b) of the *Pharo* IDE for the method `withAll:`. Developers use these two UIs to browse references of methods. The structure of these two UIs is similar: The top part provides a list of all the senders (or implementors) of a method, while the bottom part lets the user read and eventually modify the code of the method selected from the list above. *Pharo* follows a *message passing strategy*. Thus, in *Pharo* jargon, the *senders* of a method are all the methods in the system that invoke that method, or *send that message*. *Implementors*, are all the methods in the system with the same name.

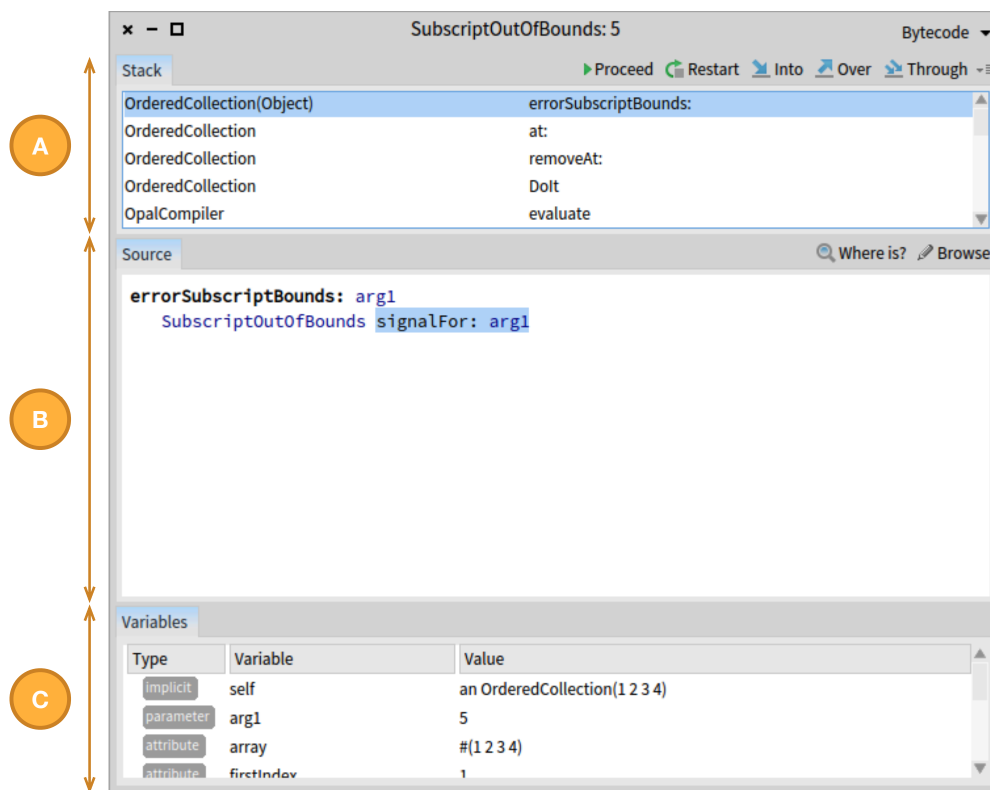


Figure 3.5. The Debugger of the *Pharo* IDE

3.4 Why *Pharo*?

We decided to carry out research with the *Pharo* IDE for a number of reasons, the first being the *"human factor"*. *Pharo* is an open-source project supported by an extremely enthusiastic user community spread between academia and industry.[5] This enabled us to stay in touch with both the user base and the core development team, situated at *INRIA* Lille,[6] a research center in northern France. In the early stage of our research, in fact, we leveraged this factor by visiting the *Pharo* team at *INRIA* to discuss our research plan and its implications on the community. This proximity enabled us to directly get in touch with people and see the impact of our research *in the wild*. Our datasets, in fact, "*are not constructed and designed with research questions in mind, as in conventional surveys, censuses, interviews, logs, observational studies, and experimental studies*" [ABSN13]. Ang *et al.* call this process "*data in the wild*".

The second reason to choose *Pharo* is the language itself, the most modern implementation of *Smalltalk*. Despite the fact that *Smalltalk* is not among the most used languages, it anticipated and inspired almost all the foundations of modern Software Engineering: from GUIs to IDEs, from test-driven development (TDD) to design patterns, from reflection to the pure object-oriented paradigm, *etc.* Programmers and language designers look at *Smalltalk* as a reference for ideal software [Chl13]. *Smalltalk* was recently appointed as the second most loved programming language[7] according to a survey on *Stack Overflow* involving more than 64,000 developers [SO17].

Smalltalk failed to dominate the world [Eng16b], but there are many reasons in favor of using

---

[5] See `http://pharo.org/Companies` and `http://pharo.org/success`
[6] See `https://www.inria.fr/en/centre/lille`
[7] % of developers who use it and have expressed interest in continuing to do so.

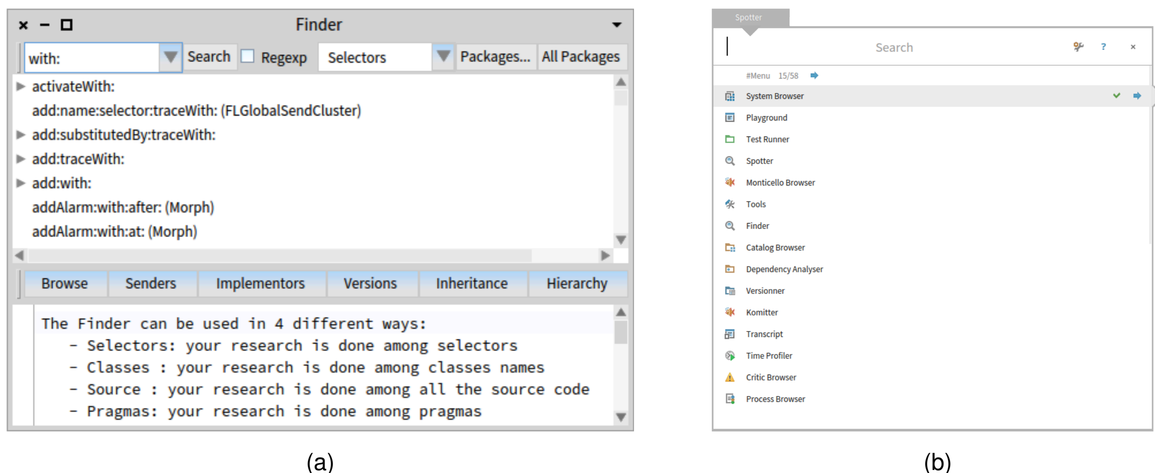


Figure 3.6. Two search user interfaces of the *Pharo* IDE: Finder (3.6a) and Spotter (3.6b)

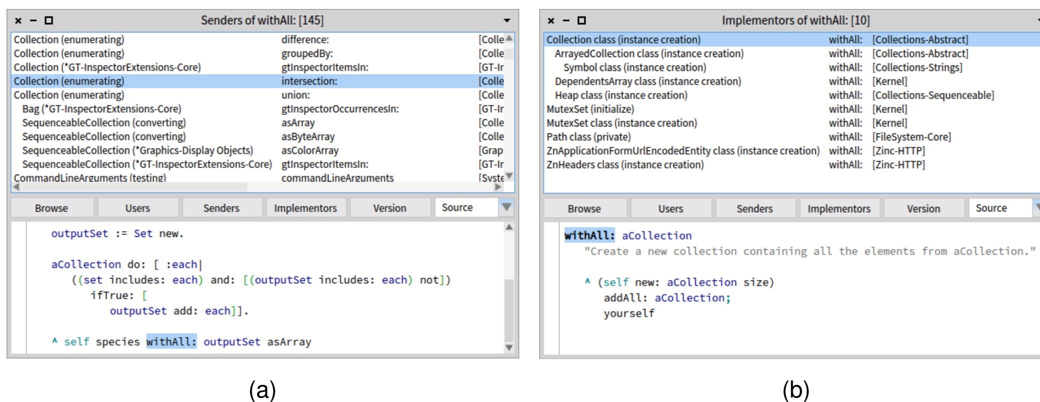


Figure 3.7. The Senders (3.7a) and Implementors (3.7b) Browsers of the *Pharo* IDE

it (*e.g.*, [Leo07], [Chl13], [Eng16a]). One for all is the fact that it is always *live*: There is no need to *start an application*, it is always up and running, source code changes have immediate effects, you can program inside the debugger, and so on. In addition, Smalltalk is simple: Its syntax (except for primitives) fits on a postcard, as shown by the following famous code snippet by Ralph Johnson, a member of the *"Gang of Four"* [GHJV95]:

```
exampleWithNumber: x
   | y |
   true & false not & (nil isNil) ifFalse: [self halt].
   y := self size + super size.
   #($a #a "a" 1 1.0)
      do: [ :each |
         Transcript show: (each class name);
            show: '']．
    ^x < y
```

Finally, *Pharo* is open-source and it is written in itself. This enables us to better understand (or alter) its behavior by reading (or changing) its source code.

3.4.1 Tab- vs. Window-based Environments

IDEs adopt one of two following UI metaphors: tab- or window-based. In a window-based IDE like *Pharo* (Figure 3.8a) multiple, potentially overlapping, windows reside under a single parent window. Other IDEs, like *Eclipse* (Figure 3.8b), instead employ a tab-based metaphor, where multiple panes are contained in the same window and navigated using *tabs*.

> *"All interaction idioms have practical limits."*
>
> — Alan Cooper *et al.* [CRC07]
> *About Face 3: The Essentials of Interaction Design.*

In our research we target the *Pharo* IDE, thus our findings are directly applicable only to this IDE. However, the two UI paradigms are essentially equivalent: A tab-based IDE is a window-based IDE with tiled windows that can be navigated by means of graphical control elements (*i.e.,* tabs). Even though this is a rather simplistic point of view, we believe that most of our approaches, and results, can be adapted to other IDEs and UI metaphors. To support this argument, in addition to *Smalltalk* interaction histories, Chapters 6 and 13 consider developer interactions with the *Eclipse* IDE, uncovering similar UI problems and findings in both IDEs.

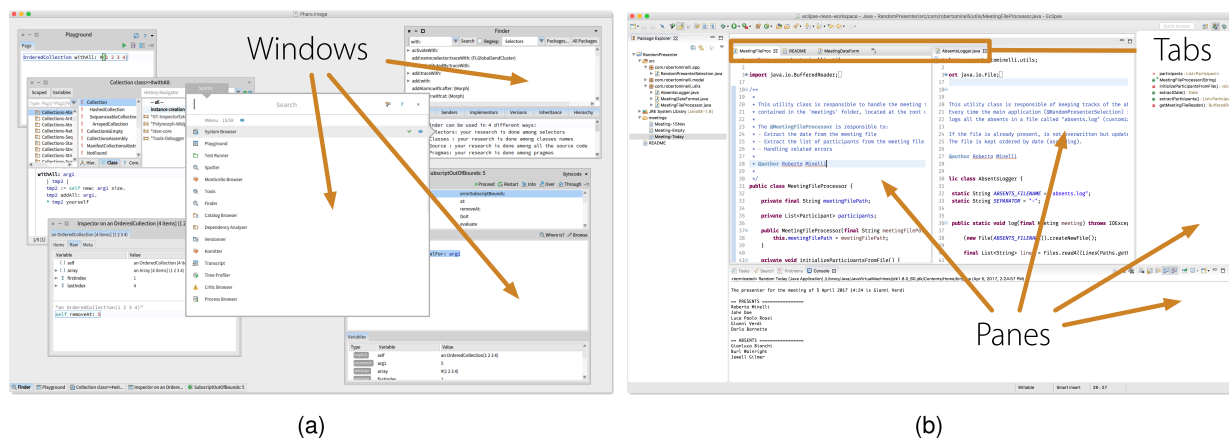


Figure 3.8. The window-based UI of Pharo (3.8a) and the tab-based UI of Eclipse (3.8b)

4

State of the Art

THIS CHAPTER discusses the state of the art in different fields related to our research. Since our work is not the first attempt to record and leverage interaction data inside the IDE, we first discuss previous approaches in this field. The main use of interaction data is to understand the behavior of developers inside the IDE. We describe works aimed at analyzing both interaction data and other sources of information. Besides interaction data, in fact, researchers also used other sources of information to analyze the behavior of developers and to provide them with additional support inside the IDE. Biometric data is one of the most recent trends. Researchers use data collected from biometric sensors (*e.g.,* heart rate, respiratory rate, electro-dermal activity) to better understand what developers perceive and how they behave during their work. Researcher also used data coming from different sources to provide developers with better support inside the IDE. In the literature the most used tools to support developers are the so-called *recommender (or recommendation) systems*. Recommender systems support users in their decision-making while interacting with large information spaces. Robillard *et al.* tailored this concept to software engineering and introduced Recommendation Systems for Software Engineering (RSSE), tools that provide information that is valuable for a software engineering task in a given context [RWZ10]. In the last years, researchers started to build recommender systems leveraging novel sources of information, such as interaction data [MFR14]. We conclude the chapter by discussing privacy and ethics issues in the context of the collection of sensible data from developers.

Structure of the Chapter

Section 4.1 lists previous approaches aimed at recording interaction data inside the IDE. Section 4.2 summarizes how researchers used different sources of information to better understand and characterize software developers. Section 4.3 describes approaches that use interaction data to support software development. Finally, Section 4.4 discusses privacy and ethics concerns related to the collection of sensible data from human subjects and discusses our experience.

4.1 Recording Software Development Data

The first development data were recorded in the late 90s through the Personal Software Process (4.1.1). Later, MYLYN opened the *Interaction Data Era* (4.1.2).

4.1.1 The 1990s: Early Development Data

In the late 90s Humphrey introduced the Personal Software Process (PSP), a software process aimed at planning, measuring, and managing the work of developers [Hum02]. As part of PSP, to augment their productivity, developers have to manually keep track of the *effort* spent on programming activities, *i.e.,* time required to complete a task. Soon researchers realized that collecting data by hand is both time consuming and error prone. For this reason they started to develop approaches to automatically gather data during software development.

PROM (PRO Metrics), is a tool that automatically collects and analyzes software metrics and PSP data [SJSV03]. It uses a plug-in architecture to collect data from development tools. HACKYSTAT is an open-source platform that uses a similar architecture to collect and analyze software development process and product data [JKA+03, JKA+04]. At regular intervals it captures activity data from the IDE and sends it to a web server. GINGER2 is a rather intrusive environment for software engineering that collects and aggregates data from multiple sources such as mouse clicks, keystrokes, eye-tracker, and skin resistance level[1] [TMN+99]. GRUMPS, instead, was developed to monitor how students use the computer for extended periods [TK03]. It records low-level actions such as mouse clicks, keystrokes, and window changes.

4.1.2 The 2000s: The Interaction Data Era

In the last two decades, researchers increasingly recognized the importance of interaction data besides PSP data. Murphy *et al.* pointed how how it can be used to evolve the environments according to user needs, to provide means to evaluate new tools and application programming interfaces, and to prevent feature bloat (*i.e.,* the tendency to add unnecessary features to a software system) [MKF06]. Frey *et al.* believe that future program investigation tools need to track the way developers navigate code to support software engineering activities [FGS11]. The most prominent tool that monitors the programmer activities inside the IDE is the MYLYN project,[2] formerly known as MYLAR [KM05]. MYLYN is a plug-in for the *Eclipse* IDE that uses the interactions to compute the degree-of-interest (DOI) of all program entities the developer interacts with. The DOI model privileges the information that developers require for the current context and helps to identify the program elements potentially relevant for the current development tasks. The authors claim that by leveraging the DOI model developers could potentially navigate the code base more efficiently. MYLYN is currently the task and application lifecycle management (ALM) framework for *Eclipse*. Murphy *et al.* used the data collected by MYLAR to study how Java developers use the *Eclipse* IDE [MKF06]. Among their findings, they discovered that developers use most of the *Eclipse* perspectives while developing, and that keyboard shortcuts are a frequently used alternative to reach some IDE features. Researchers showed that MYLYN data has limitations [VCN+11, YR11, SRG15]. The granularity of the data, for example, is very coarse: Events are aggregated making it hard (or impossible) to know the exact sequence of interaction events as they happened in reality [YR11]. Driven by the limitations of existing recording

[1]This is also known as Electrodermal Activity (EDA) or Galvanic Skin Response (GSR).
[2]See `http://www.eclipse.org/mylyn`

tools, Vakilian *et al.* developed CodingSpectator and CodingTracker [VCN+11]. These tools are aimed at collecting rich data about high-level refactorings and low-level code edits.

Researchers also extended and customized Mylyn and its functionalities. Fritz *et al.*, for example, refined the DOI model of Mylyn introducing a Degree of Knowledge (DOK) model [TFMH10]. This model measures who has more familiarity with a particular source code element by considering both authorship information and developer's interactions with the code. Kobayashi *et al.* developed Plog, an extension of Mylyn, that captures interaction histories inside the *Eclipse* IDE [KKA12]. Plog records interactions at both file and method level. Plog distinguishes actions that *modify* the source code and actions that only *reference* it. Differently from Mylyn, the authors use interaction histories to generate a change guide graph to support file- and-method level change prediction.

Coman and Sillitti tracked the sequence of methods developers interact with and used this data to automatically split development sessions into task-related sub-sessions [CS08]. Yoon and Myers proposed FLUORITE, an event logging plug-in for the *Eclipse* IDE that records low-level events in the code editor [YM11]. The tool logs three types of events: commands (*e.g.,* copy-paste, typing text, moving the cursor), document changes (*i.e.,* whenever the active file is changed), and annotations. The purpose of the tool is to evaluate existing tools through the analysis of the coding behavior of developers. Among their findings, the authors provided empirical evidence that editing source code is different from editing textual documents. The authors also studied the distribution of keystrokes reporting that *backspace* and *arrows* are the most frequent keys pressed by developers.

Gu *et al.* developed IDE++, a plugin for the *Eclipse* IDE [GSBS14]. IDE++ is an extension of Mylyn that captures more fine-grained interactions with respect to Mylyn.[3] As a proof-of-concept, the authors built different applications on top of IDE++, *e.g.,* test case recommendations, summarization of a development session.

Most of the research on interaction data targets the *Eclipse* IDE and it is related to the Mylyn project. Researchers, however, also developed approaches for other IDEs. Snipes *et al.* developed Blaze, a tool that introduces game design elements to improve navigation practices of developers inside the Visual Studio IDE [SNMH14]. During the experiment, the authors used Blaze to record usage data from six developers. Damevski *et al.* also recorded and studied interaction data inside the Visual Studio IDE [DCSP16, DSSP17]. The authors collected a large-scale dataset of IDE interactions from more than 200 industrial developers working for ABB.[4] The authors made their dataset publicly available for researchers [ABB17]. Amann *et al.* also developed a plugin to collect interaction data in the Visual Studio IDE [APNM16] and used the recorded data to better understand how developers use this IDE.

4.2 Understanding the Behavior of Developers

Researchers studied different aspects of the behavior of software developers. For example, they studied how developers navigate source code, how fragmented is their work, or how difficult is to create and maintain the context in a development session. To do so, they leveraged different information sources such as interaction data, fine-grained source code changes, versioning system data and, in more recent times, also biometric data such as heart rate and electrodermal activity.

[3] Unfortunately, we did not succeed in installing the tool on three different versions of *Eclipse*.
[4] See `http://abb.com`

4.2.1 Understanding Source Code Navigation

An essential activity for developers is navigating the software at hand. According to Wexelblat and Maes, the information path obtained from navigation in an information space reveals the user's mental model of the system [WM99]. In software engineering a mental model is *"a link between the representation in the mind of a developer and the source code"* [FGS11]. Constructing and maintaining mental models are essential activities to support the understanding of programs [SLVA97, Wal03, RCM04, KMCA06].

Navigating source code is challenging because of the complex nature of code: The relevant code fragments are often dispersed in several locations in the system. Kersten and Murphy argued that *"developers tend to spend more time navigating the code than working with it"* [KM05], which throws up the question whether IDEs appropriately support the navigation. Ko *et al.* conducted an observational study to understand how developers gather information that are necessary to make changes to a software system [KMCA06]. They found that, on average, developers edit unfamiliar source code for a fifth of the time and reported that developers spend 35% of their time navigating the source code in search for information. In addition, 27% of the navigation operations are performed on already visited locations, indicating the necessity to periodically revisit these locations to recall information no longer visible on screen. The authors reported that often developers perform *back-and-forth navigations* between different files to compare different pieces of code. This is an indication that the IDE might not sufficiently support the navigation, *i.e.,* only one tab is visible at a time.

Zou and Godfrey provide additional evidence on the importance of navigating software artifacts [ZG06]. The authors conducted an industrial case study to understand which program artifacts are viewed during maintenance tasks. In more than 70% of the cases, the number of viewed-only artifacts is larger than the number of modified artifacts [ZG06]. Also according the analysis of development interaction logs conducted by Snipes *et al.* developers spend more than half of their time browsing and reading source code inside the IDE [SNMH14].

Soh *et al.* conducted another study to discover how developers explore software systems during maintenance tasks [SKG+13]. They characterized the type of exploration as either *referenced* or *unreferenced*: Referenced exploration means that the developer often revisits one or more entities with higher frequency with respect to the others, while unreferenced exploration implies that developers visit all the program entities with almost the same frequency. Among their findings, they discovered that during maintenance tasks developers mostly follow unreferenced exploration, *i.e.,* they visit all the entities with the same frequency.

Piorkowski *et al.* conducted different studies on source code navigation [PFS+11, PFK+13]. The literature offers a plethora of predictive models to support source code navigation. Piorkowski *et al.* compared a substantial amount of models to assess their predictive accuracy [PFS+11]. Consistent with prior work [PG06], they discovered that *"recency"* was the most accurate model to predict click-based navigations. However, there is not a single model that is good at predicting different kinds of navigation. For this reason they suggest that we should combining multiple single-factor models to predict programmer navigation more accurately. Singh *et al.* partially replicated the study of Piorkowski *et al.* [PFS+11] and compared different models of programmer navigation [SHFL16]. Click-based navigation is the model that better records a developer's navigation behaviors with respect to the view-based model. Consistent with the results of Piorkowski *et al.*, the predictive model based on *"recency"* outperforms all the models. In another study Piorkowski *et al.* observed what programmers need during debugging tasks and how they forage to satisfy their needs [PFK+13]. Among their results they discovered that participants spent 50% of their time foraging for information, highlighting the importance of

source code navigation in software development. Interestingly, they also discovered that different developers have different needs and foraging techniques to accomplish the same task.

Fritz *et al.* conducted an exploratory study to understand how developers create context models needed to perform change tasks [FSK+14]. Among their results, consistent with the results of Piorkowski *et al.* [PFK+13], they observed that navigation strategies differ substantially between different developers. Moreover, they report that developers use a combination of search and navigation to explore source code.

4.2.2 Understanding the Role of Program Comprehension

Inside the IDE, developers mainly read and write source code. It has been shown that developers spend more time reading source code than writing it [VMV95]. Reading code is the foundation of program understanding, which has been estimated to occupy half of the work time of developers [FH83] and to be one of the most challenging tasks performed by developers [LVD06]. To read code, developers have to explore and navigate the system at hand [KM05] to build their mental models of the system [KMCA06, RCM04, SLVA97, Wal03], *i.e.,"a link between the representation in the mind of a developer and the source code"* [FGS11].

In the last 40 years, program comprehension (or program understanding) has been the target of many empirical and observational studies. The cost of software comprehension—that includes the time required to understand it and time lost in misunderstanding—is rarely seen as a direct cost, but is significant [CC90]. Zelkowitz *et al.*, for example, estimated that program comprehension takes more than half the time spent on maintenance [ZSG79]. In turn, according to Erlikh, maintenance accounts for 55% to 95% of the total costs of a software system [Erl00], thus the weight of program comprehension globally ranges between 30% and 50%. This estimation is also corroborated by Fjeldstad and Hamlen, who claim that comprehension occupies half the time of developers [FH83].

LaToza *et al.* surveyed more than 150 developers from Microsoft Corporation to identify the issues they encounter during source code comprehension [LVD06]. More than half of the respondents agreed that the most serious problem is understanding the rationale behind a piece of code. In particular, developers wonder why the code is implemented the way it is or what it is trying to achieve. LaToza *et al.* reported that developers often are disoriented when they have to deal with unfamiliar source code. According to Ko *et al.*, developers have to deal with unfamiliar code more than 20% of the time [KMCA06]. In addition, Ko *et al.* claim that code understanding is also achieved by navigating source code fragments, and that navigation occupies around 35% of the total development time [KMCA06].

Singer *et al.* studied how developers use their time, mainly by using questionnaires [SLVA97]. Although they did not use a precise measure for the time taken by developer activities, they noticed that the time spent on writing source code is less than the time spent on other activities, like debugging or searching.

4.2.3 Understanding Tasks and Work Fragmentation

During development, programmers construct and maintain the context of entities relevant for the current task. Since this process is non-trivial, researchers proposed various approaches to improve the construction and management of working sets. Work fragmentation, interruptions, and context switches add another level of complexity to this process. For this reason, researchers studied how interruptions impact the workflow of developers and devised approaches to recover the context after interruptions.

Sillito *et al.*, for example, studied which questions programmers ask themselves when evolving a code base [SMDV08]. They identified 44 kinds of questions grouped into 4 categories: Finding initial focus points, building on those points, understanding a subgraph, and questions over groups of subgraphs. To answer the majority of these questions, developers have to consider one or more subgraphs of the system, significantly increasing the working set size.

In a single development session developers often work on multiple tasks. Researchers proposed techniques such as concern inference [RM03] and concept identification [CG07] to assist developers in recovering the contexts after a switch. However most approaches rely on developers to manually indicate the beginning and the end of a task [CS08]. Coman and Sillitti used a single interaction event, the change of the method in focus, to automatically identify different tasks in a development session [CS08]. In their pilot study, they correctly identified more than 80% of tasks.

Murphy *et al.* observed that most of the development task carried our by developers have a *structure* that emerges from how a developer works with the code base [MKRČ05]. The authors defined the task structure as *"the parts of a software system and relationships between those parts that were changed to complete the task"* [MKRČ05]. Murphy *et al.* believe that the task structure can be used to enhance the collaboration between developers inside the IDE [MKRČ05].

Ying and Robillard analyzed MYLYN interaction histories and BUGZILLA[5] bug reports to characterize the editing behavior of developers [YR11]. They identified three main editing styles: edit-first, edit-last, and edit-throughout. They observed that the editing behavior of developers is correlated with the type of task performed, *i.e.,* enhancement tasks, minor, and major bug fixes. Among their results they discovered that in enhancement tasks, as opposed to bug fixing tasks, developers tend not to follow an edit-first style. According to Ying and Robillard, IDEs can track the editing behavior of developers to customize the UI of the IDE, *i.e.,* in an edit-first session the IDE can show more editing-related features than instead of navigation-related tools.

Researchers also studied the size of the context model (or working set) needed to perform a development task. Fritz *et al.* conducted two observational studies of 12 developers, each solving three tasks, to understand how big is the context model needed to complete a change task [FSK⁺14]. Among their results, they discovered that on average the context model necessary to solve a task contained 4 classes. However, different developers have different needs, thus the size of context models can vary substantially between developers. Researchers also correlated the size of the working sets with different efficiency measures such as code completion [HKR⁺14, EHRS14, PHR14]. Hanenberg *et al.* investigated the influence of type systems on software maintainability by measuring the number of files opened and file switches [HKR⁺14]. They observed that developers using statically typed languages tend to open less files and switch less often between them. In addition to static type systems, researchers also observed that textual documentation helps developers to reduce the completion time of a task, *i.e.,* smaller working sets and less files switches [EHRS14, PHR14].

The creation and management of working sets is strictly influenced by the amount of fragmentation in the workflow of developers [MGH05, ZG06, KDV07, SRG15]. Zou and Godfrey investigated which program artifacts are viewed during maintenance tasks [ZG06]. They collected interaction histories from developers and observed that there are periods of time where there is no activity. They discovered that, most of the times, this periods of inactivity are indeed interruptions, *i.e.,* phone calls.

Ko *et al.* analyzed the information needs of software developers [KDV07]. They discovered that, on average, developers are interrupted every 5 minutes, consistent with the results of Mark

[5] See `https://www.bugzilla.org`

et al. [MGH05]. Most of the interruptions were due to face-to-face communication, instant messaging, or phone calls. Notifications, such as email and alerts, are another important reason for fragmented work. Another factor contributing to context switches is the need for developers to find knowledge outside the IDE (*e.g.,* when learning to code with a new API).

Sanchez *et al.* studied MYLYN data to understand if work fragmentation has an impact on the productivity of developers [SRG15]. They observed that the productivity decreases with the increase of the number and the duration of interruptions in the developer workflow.

4.2.4 Leveraging Fine-Grained Source Code Changes and Biometric Data

Besides interaction data and observational studies, researchers also leveraged other sources of information to capture the behavior of software developers. Fine-grained source code changes and biometric data are the most leveraged sources of information.

Robbes and Lanza, for example, proposed SPYWARE a change-aware development toolset that records and leverages fine-grained source code changes [RL07, RL08]. Different from state-of-the-art versioning systems, fine-grained source code changes capture all the effective modifications that a developer performs on a software system [RL07]. The authors devised ad-hoc metrics such as *Total Number of Changes*, *Session Activity*, and *Session Focus*, and used them to characterize development sessions [RL07]. In particular, they used a visualization to identify four kinds of development sessions: i) decoration, ii) masonry, iii) painting, and iv) architecture & restoration. On top of SPYWARE, Robbes and Lanza built approaches to improve code completion [RL10] and to enhance existing change prediction approaches [RPL10].

Negara *et al.* mined fine-grained sequences of code changes to detect *previously unknown* code change patterns [NCDJ14]. Among the different kinds of program transformations discovered by the authors, participant reports the following three to be the most relevant: i) *Changing a Field Type*, ii) *Creating/Initializing a New Field*, and iii) *Adding Precondition Checks for a Parameter.*

Fritz *et al.* used biometric data to assess the difficulty of code comprehension tasks [FBM+14]. They combined data coming from an eye-tracker, an electrodermal activity sensor, and an electroencephalograph to predict task difficulty. The proposed approach can predict whether a developer will perceive her tasks as difficult with 70% of precision. Fritz and Müller investigated how to leverage biometric data to boost the productivity of developers [FM16]. In particular, they used various biometric measurements to sense i) task difficulty, ii) progress and emotions, and iii) interruptibility with good levels of precision and recall in both lab and field settings.

4.2.5 Visualizing Software Development

Software visualization is a specialization of information visualization that focuses on software [Lan03]. Stasko *et al.* defined software visualization as "*the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software*" [SDBE98]. Researchers categorized visualizations according to their intended scope (descriptive, analytical, or exploratory [BAB+93]) and the type of data visualized (algorithm animations, dynamic, and static visualizations [PBS93]).

Researchers used software visualization as means to support the understanding of different aspects of software development. Gîrba *et al.* visualized code ownership with the *Ownership Map* view [GKSD05]. The authors defined a measure of code ownership and then build a visualization to understand when and how different developers interacted with a system. The *Ownership Map* uncovered several behavioral patterns of developers during the evolution of software systems.

For example, the authors identified *monologue* periods, in which a single developer makes most of the changes, *teamwork* periods where two (or more) developers commit a quick sequence of changes to multiple file, and *silence* periods with nearly no changes at all. Figure 4.1 shows an example of the ownership map that exhibits different behavioral patterns such as a *monologue* of the Green author and a *familiarization* period of the Blue author.

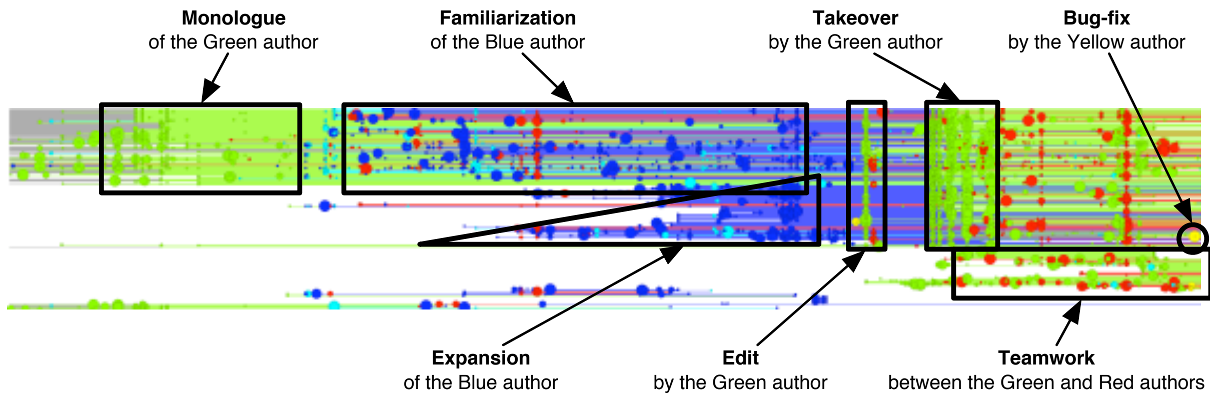


Figure 4.1. An example of the Ownership Map exhibiting different behavioral patterns

Greevy *et al.* visualized code ownership in a structural fashion [GGD07]. They propose a *Package Owner* view to depict ownership information for the entire system and a *Team Collaboration* view that shows how developers collaborate between themselves to develop features.

Telea and Auber developed CODE FLOWS, a set of visualization techniques to analyze the evolution of source-code structure [TA08]. CODE FLOWS visualizes code correspondences using *textured splines connected to mirrored icicle plots.* Figure 4.2 shows how CODE FLOWS visualizes code evolution: from source code, to trees of matching correspondences, to the visualization.

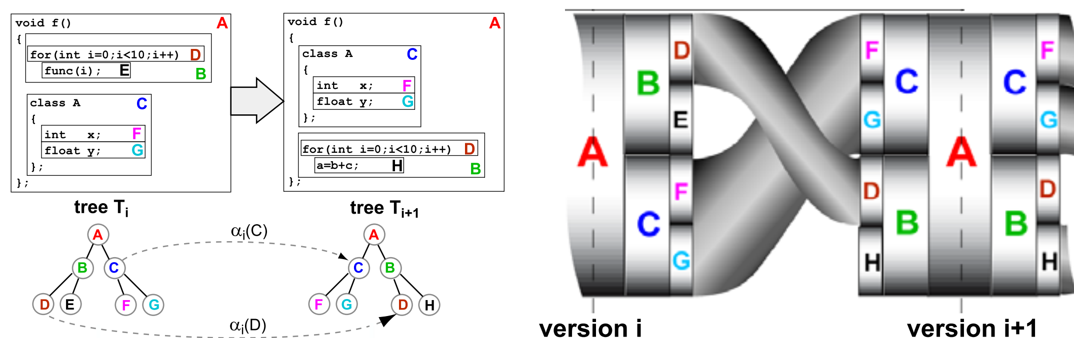


Figure 4.2. Code Flow: From the source code to the visualization

This technique enables to keep track of unchanged code and to detect and highlighting important events such as code drift, splits, merges, insertions, and deletions.

Ogawa and Ma propose two views of source code and developers: CODE_SWARM, a tool that produces animated software histories from data coming from version control systems [OM09] and a historical visualization to show the interactions between developers in the evolution of software projects [OM10]. Figure 4.3 shows an example of the CODE_SWARM visualization.

Besides trying to understand the behavior of software systems and software developers, researchers also used software visualization to support software development inside the IDE.

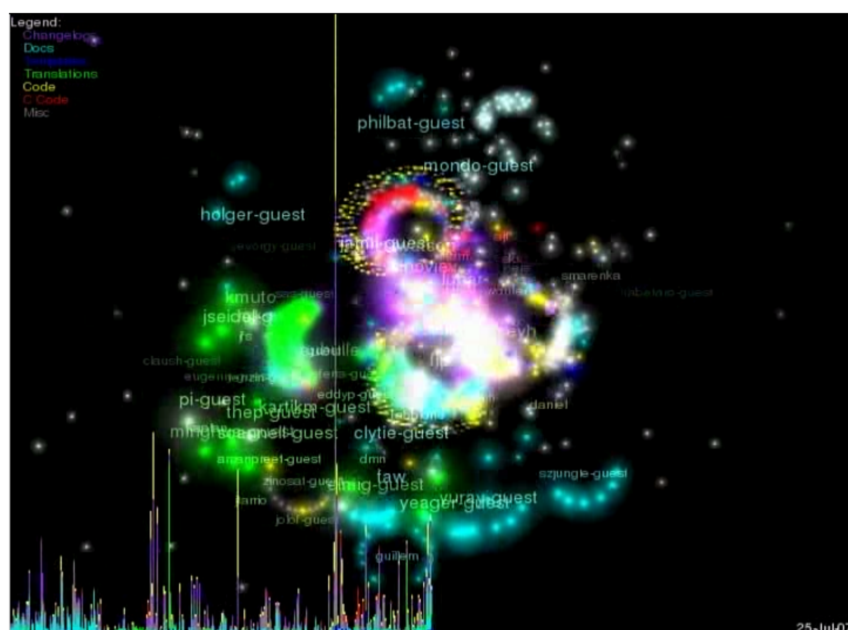


Figure 4.3. The code_swarm visualization: an experiment in organic software visualization

Desmond *et al.*, for example, developed FLUID source code views, a novel source code document presentation [DSE06]. FLUID allows programmers to work on a primary source code file and later to *fluidly* shift focus to related material based on the development context. According to the authors, this approach provides better support to comprehension tasks and reduces the time spent in navigating between software artifacts.

Yoon *et al.* developed AZURITE, an *Eclipse* plug-in that visualizes fine-grained code change histories [YMK13]. The tool provides a "*timeline*" that lets developers navigate the history of changes and quickly reach the needed information. AZURITE also offers a "*code history diff*" to inspect the changes of particular code fragments.

4.3 Supporting Software Development Activities

Besides using software development data to better understand the behavior of software developers, researchers also used this data to provide better support to software development activities.

4.3.1 Supporting the (Re)construction of Working Sets

Developers are frequently interrupted during their daily work. Recovering from interruptions can be difficult [LVD06]. Face-to-face chats, emails, or instant messages are only a few causes for a very fragmented workflow. Researchers showed that, when interrupted at the wrong moment, developers require more time to more time to complete the tasks, they commit twice the number of errors across tasks, and experience more annoyance and anxiety [BK06]. For this reasons, researchers developed approaches to support the recovery of the development context after an interruption or to identify a suitable moment to interrupt a developer during her workflow.

MYLYN itself is a tool that improves the management of the working set by assigning a degree-of-interest (DOI) to program entities [KM05]. The tool features an episodic-memory inspired interface that filters our the entities with a low DOI that allows developers to only

focus on the entities they require for the current task. This reduces the information overload and potentially decreasing the effort needed by the developer to recover her previous context. Kersten and Murphy empirically demonstrated that the task context (and DOI model) of MYLYN can improve the productivity of developers [KM06].

Röthlisberger *et al.* proposed SMARTGROUPS, a tool that keeps track of navigation and edit activities and combines this data with evolutionary and runtime information to provide developers with a more structured view of the entities potentially needed to complete the current task [RND11]. The tool also provides a suggestion list of the artifacts that might be relevant for the task at hand. The authors showed that with SMARTGROUPS developers spend less time navigating the software space.

In their work, Züger and Fritz used psycho-physiological sensors information to predict the impact of interruptions during the workflow of developers [ZF15]. With their approach they were able to automatically identify with high accuracy, both in the lab and in the field, the most suitable moments to interrupt a knowledge worker, mitigating the side effects of interruptions. On top of this study, Züger *et al.* developed FLOWLIGHT, an approach that uses a physical traffic-light like LED to signal when it is more suitable to interrupt developers [ZCM+17].

Parnin and Görg used developers interactions with the IDE to define the development context as the set of methods potentially relevant for a task at hand [PG06]. According to the authors, recommendation systems can use this context to support the recovery of the mental state of a developer and to facilitate the exploration of software systems [PG06].

Separation of concerns in software systems is not always optimal. To support developers in modifying concerns that are not well modularized, Robillard and Murphy introduced *concern graphs* [RM07]. Concern graphs are artifacts that explicitly document the implementation of a concern, *i.e.,* capture the parts of the system that are relevant for a given concern. Their tool, FEAT supports developers in constructing concern graphs semiautomatically as part of their normal program investigation activities. At a later time, developers can use the tool to focus on the parts of the system that are relevant for the current concern (or task).

4.3.2 Supporting Source Code Exploration and Navigation

Source code navigation is one of the main activities performed by developers in the IDE [KMCA06, MML15b]. Researchers developed various approaches to support this activity inside the IDE.

Storey *et al.*, for example, developed SHRIMP: A flexible and customizable environment to visually explore software systems [SM95, SBM01]. SHRIMP offers a catalog of graph-based architectural visualizations that combine data from different sources to provide a more structural exploration of code. The authors claim that by embedding code and documentation in the same view, SHRIMP provides developers with quick and easy support to construct their mental models.

Janzen and De Volder proposed JQUERY, a tool that combines the advantages of structural source code browsers and query based tools to reduce the confusion while navigating code [JD03]. Their approach provides an explicit representation of the exploration process by means of *exploration paths*. The users of JQUERY reported that *exploration paths* are helpful to keep the focus during an exploration task.

Singer *et al.* devised a methodology backed up with a tool called NAVTRACKS that monitors the navigation histories of developers to support browsing through software [SES05]. When a developer selects an entity, NAVTRACKS shows a list of artifacts potentially related to it. NAVTRACKS models relationships between files and entities and leverage these relationships to recommend potentially related files as a developer navigates the system. The tool also provides a graph based visualization to depict how program entities are related between themselves.

Team Tracks is also a tool that monitors the interactions of the developer with the IDE to support navigation through software [DCR05]. The main difference with NavTracks, as the name suggests, is the fact that Team Tracks also leverages the navigation histories of the other members of the development team. The tool provides a view to browse the items related to a particular entity and a *Favorite Classes* view showing a class hierarchy view with only the most frequently visited elemets (*i.e.,* classes, methods, and members). As a result of the evaluation of Team Tracks, the authors claim that sharing navigation data within a development team can ease program comprehension tasks.

Chronos is an *Eclipse* plug-in that lets developers visually query and explore historical source code change events [SJ13]. The authors claim that by answering questions about source code history, Chronos augments developers' productivity.

Augustine *et al.* investigated how to comprehend and maintain source code more efficiently by fostering structural code navigation [AFQ+15]. The authors developed Prodet, a tool that provides a navigable call-graph visualization of the relevant parts of the call graph based on the current context. In their study they show that the interactive visualization increases the effectiveness of developers in navigating source code elements.

4.3.3 Towards the Next Generation of IDEs

Software development data, such as IDE interactions, can be used to understand and support software developers. The next step—as envisioned by Murphy *et al.*—is to evolve development environments according to user needs [MKF06]. Developers spend much of their time reading and analyzing code and mainstream IDEs are essentially glorified text editors mostly treating source code as text [Nie16]. Building on this statement, researchers proposed approaches to improve the user interfaces (UIs) of current IDEs or invented new IDE paradigms.

Researcher showed that the UIs of current IDEs have significant limitations. They proposed approaches and tools to mitigate such problems. Rötlisberger *et al.*, for example, observed that developers are typically confronted with a large number of windows (or tabs) inside the IDE, most of which are irrelevant for the current development session [RND09]. They called this phenomenon *window plague* and they propose AutumnLeaves to mitigate this problem [RND09]. AutumnLeaves is a tool that keeps track of the importance of UI elements visible on screen (*i.e.,* windows or tabs) and gently removes the ones that are less likely to be used again in the future. The tool automatically suggests which windows or tabs can be closed to reduce the amount of noise present in the IDE.

Lee *et al.* claimed that current support provided by IDEs to refactor source code is inefficient [LCJ13]. To provide better refactoring support, they introduced "*Drag-and-Drop Refactoring*" in the *Eclipse* IDE [LCJ13]. In their evaluation they discovered that their approach is intuitive, more efficient, and less error-prone compared to the refactoring capabilities of current IDEs.

Researchers claimed that IDEs should present the information following the mental models of programmers [BRZ+10, DR10, OLDR11]. In one of our studies, we showed that developers spend a non-trivial amount of time spent in fiddling with the UI of the IDE that calls for research on novel UIs and interaction paradigms [MML15b]. In the last decade, researchers have also investigated better program representations and UI paradigms than the file-and-tab-based metaphor of most common IDEs. We can trace back this inspiration to the *Lisp* and *Smalltalk* IDEs of the 80's, whose most recent representative is *Pharo*. Two notable examples of alternative IDEs are Code Bubbles [BZR+10, BRZ+10] and Code Canvas [DR10]. The aim of these tools is to reduce the amount of time spent in navigating the system at hand by maximizing the number of entities visible at the same time.

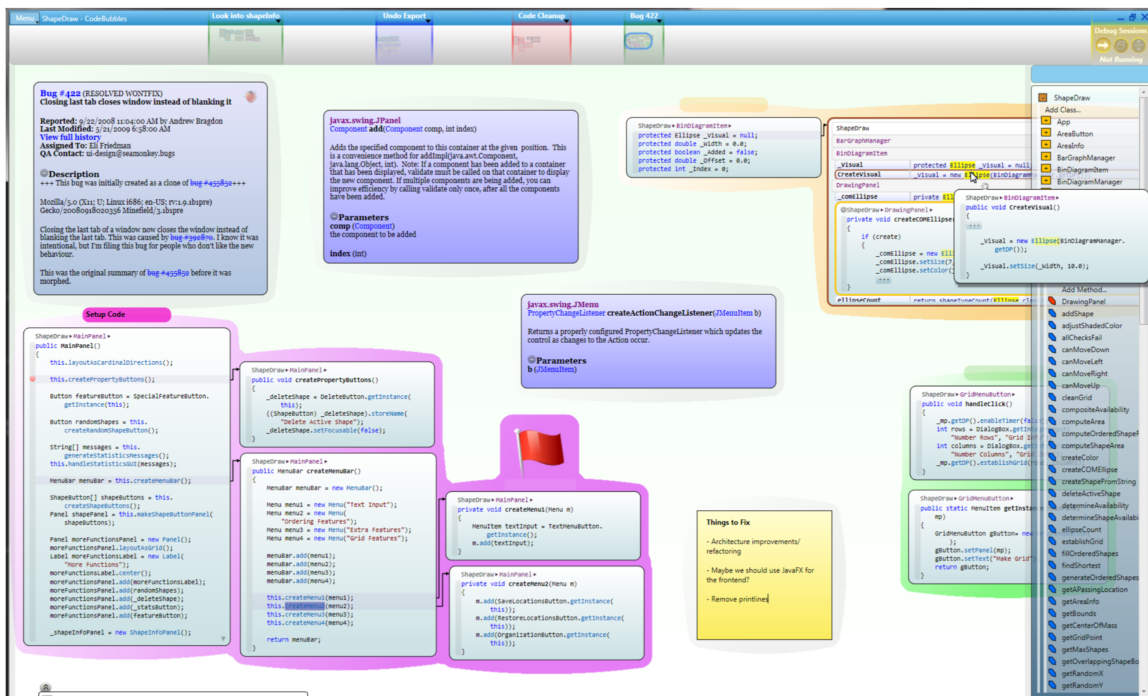


Figure 4.4. A screenshot of Code Bubbles

Figure 4.4 displays a screenshot of Code Bubbles. The tool is a font-end to the *Eclipse* IDE and enables developers to create and work with working sets instead of standard Java files. A working-set is a group of methods, documentation, notes, and other pieces of information that the developer considers relevant for the task at hand. Instead of working with Java files, in Code Bubbles developers interact with compact function-based views of the code called *bubbles*. In the evaluation of the tool the authors showed that at a similar screen resolution Code Bubbles was able to show more methods at the same time than the classic *Eclipse* view [BZR+10]. Furthermore, a controlled experiment showed that Code Bubbles users were both more successful and faster in completing maintenance tasks than *Eclipse* users [BRZ+10]. Parts of this performance increase is attributable to a reduction of repeated navigations, such as the ones observed by Ko *et al.* [KMCA06], according to the videos recorded during the controlled experiment (75.9% of all *Eclipse* navigation operations, compared to 37.6% for Code Bubbles), as more entities were visible on screen. The benefits of Code Bubbles were studied in the context of a controlled experiment, with strong internal, but limited external validity [SSA15].

DeLine and Rowan proposed Code Canvas, an interface similar to Code Bubbles, that provides an infinite zoomable surface for software development [DR10]. Code Canvas is a front-end to the *Visual Studio* IDE that tries to overcome the "*bento box design*" of today's IDEs that partitions information into separate areas. Code Canvas replaces the bento box structure with a *canvas*: A single zoomable surface that accommodates all the elements useful for the task at hand, *e.g.,* source code, UI designs, images, debugger stack traces, and search results.

A collaboration from the teams behind Code Canvas and Code Bubbles originated a tool for *Visual Studio* called Debugger Canvas [DBR+12]. This tool is an industrial implementation of the Code Bubbles paradigm specialized for debugging activities. Debugging is a cognitively intense and navigation-heavy activity that can intensively take advantage from the the bubbles design. From their experience, the authors report that the canvas paradigm is best

embodied as a mode within the existing user experience rather than a replacement.

Another emerging idea for development environment is represented by web integrated development environments[6] such as CLOUD9[7] and CODIO.[8] Web IDEs provide the same functionalities of desktop IDEs (*e.g.,* syntax highlighting, error checking, plugin and file management) but can be accessed from anywhere. The majority of Web IDEs encourage collaboration between developers by allowing multiple developers to work in real-time on the same project. Currently they employ the same UI of desktop IDEs, but their underlying architecture is very promising.

These are only the first steps towards the next generation of IDEs. We envision IDEs that exploit the full potential of interaction data while the developer is programming to support her workflow [Min14]. For example, we foresee user interfaces that adapt their shape to the workflow of the developer by, for example, rearranging frequently used UI components such as code browsers or menus, or recommender systems that efficiently support repetitive tasks in the workflow of developers by harnessing past interaction histories.

4.4 Privacy and Ethics

Similar to many of the related works, our research relies on the collection of potentially sensible data from human subjects. In this section we overview privacy and ethics concerns raised by the collection of data from human beings and discuss our experience.

Privacy and ethics concerns in the context of human subject experiments have been widely discussed in the literature, *e.g.,* [LB12]. In his Ph.D. dissertation, Langheinrich discusses the various facets of both privacy ad ethics [Lan05]. Among the different subareas of Ethics, Applied Ethics address practical and concrete questions. This is relevant in the context of technological advancements, such as computer science: Technology creates new possibilities for human action but raises novel ethical issues [Joh99]. To support their members, professional associations such as the Association for Computing Machinery (ACM),[9] publish the so-called "*Code of Ethics*". These guidelines combine deontological ("*be honest and trustworthy*") and teleological ("*contribute to society and human well-being*") principles [Lan05]. Recently, ACM updated "The Code" to reflect the significant changes to the profession of computing happened in the last 25 years, the date of its last update [Wol16].

> *"New technologies arise so quickly that they may be in widespread use before practitioners can see the social and ethical consequences."*
>
> — BRINKMAN *et al.* [BGMW16]

4.4.1 The Case of Interaction Data

In the context of development data collected inside IDEs, Snipes *et al.* recently published a practical guide to analyze IDE usage data that also discusses about specific privacy and ethics concerns in the field [SMHF+15]. In this context, privacy concerns arise because the data collected may expose developers or parts of proprietary source code [SMHF+15]. To mitigate this problem,

[6] Also known as Web IDE, WIDE, or Cloud IDE.
[7] See https://c9.io
[8] See https://codio.com
[9] See https://www.acm.org

often researchers encrypt sensitive information such as the name of the developer, the name of the file she is working on, and the identifiers contained in the source code.

Obfuscating data is a double-edged sword: On the one side it reduces privacy concerns but on the other side it limits what you can learn from the data. A good tradeoff is using a one-way hash function, that allows to differentiate between distinct developers and identifiers, in an anonymous fashion [SMHF+15], *i.e.,* without knowing the name of developers and identifier names. Related to that, anonymity of the data prevents researchers from drawing conclusions about the data. Researchers might formulate hypotheses that require to be verified with the developers, the only subject that knows exactly the rationale behind her interactions. Anonymous data makes this step impossible. To mitigate this risk, at the beginning of the data collection process, researchers should provide subjects with a statement explaining who will access to the data and what they will do with it [SMHF+15]. To reduce privacy concerns, often researchers avoid to consider data at individual level but rather draw conclusions on the entire dataset.

> *"No one spends money collecting these data to actually learn anything about you. They want to learn about people like you."*
>
> — Lawrence Lessig [Les99]

In this context, another tradeoff lies between research risk and benefits. From an ethical perspective, if the research has more potential benefits than risks it is permissible [Tay94, Chr08].

4.4.2 Our Experience with DFlow and the *Pharo* IDE

At the beginning we were the only users of DFlow, thus we were not concerned about privacy and ethics. In a subsequent moment, when we extended the data collection to our research group. Members of our group were developing open-source software to support their research. Thus, also in this case, we were not much concerned with privacy issues.

The last step was extending our data collection to the *Pharo* community through open calls in the `Pharo-dev` mailing list, a mailing lists that includes the most active members of the *Pharo* open-source community. We invited them to voluntarily participate in our study (*i.e.,* informed consent) and set up a small website[10] to explain them how to participate in the data collection process. The website features a disclaimer saying that "*all the collected data will be treated confidentially and used only for scientific research purposes.*" Even after having agreed to participate in our study, developers can disable DFlow at any moment with a dedicated switch.

In the context of an open-source community—like *Pharo*—one might expect that developers are in favor of having tools that collect usage data with the aim of improving their development environment. However, we learned that there are two kinds of developers: hippies and paranoids. The former do not care about which kinds of data researchers collect and why and join the experiment because to help researchers. The latter, instead, want to know exactly the types of data collected and the exact purpose of the data collection. The full list of data collected is easily accessible (see Table 5.1) and our disclaimer guarantees that we will use the data only "*for scientific research purposes*". However, the experimental nature of our research prevents us from knowing the exact purpose of the data collection process. For this reason, we had to establish a relation of *trust* between us and the subjects using DFlow.

[10] See `http://dflow.inf.usi.ch/experiment.html`

Privacy Settings: More Control on Usage Data Collection in the *Pharo* IDE

With the increase of approaches aimed at the collection of development data inside the IDE, in 2016 *Pharo* introduced the global "*privacy settings*". Essentially, it is a switch that developers can toggle to help the *Pharo* community to improve its products and services by automatically sending diagnostic and usage data. Until now only a few tools rely on this, but we believe that in the near future all tools aimed at data collection will conform to it. This represents a form of *informed consent*, where individuals explicitly choose if and when to participate. For example, developers can disable the data collection when working on proprietary software and enable it when contributing to the open-source code base of the *Pharo* IDE itself. The unified switch is a good starting point, but we believe that more can be done in this direction. A possibility is that developers can choose the desired *level of privacy* in the settings, as follows:

- **High-Privacy:** All data are sent in anonymized form;
- **Mid-Privacy:** Only identifier names are sent in clear, *i.e.,* no source code;
- **Low-Privacy:** All data are sent in clear, including source code.

This enables developers to have full control on which data is being collected by researchers. For each task, developers can choose whether they do not intend to share source code (*i.e.,* they are working on a commercial product) or if they do not need obfuscation and privacy at all.

4.5 Reflections

We strongly believe that interaction data is a fundamental source of information to analyze and support the behavior of software developers inside the IDE. In our long term vision IDEs should be "*Interaction-Aware*", meaning that they should collect, mine, and leverage the interactions of developers with different information sources to support their workflow.

> *"If the last two decades could be labeled the era of big data collection, the next two decades will surely be labeled as the era of smarter big data analysis."*
>
> — W. Snipes *et al.* [SMHF+15]

The first step, monitoring interaction data inside the IDE, is shared with most of the related works discussed in Section 4.1. However, there are two main differences between previous work and our research: i) The *quality* of the recorded data and ii) the purpose for recording this data.

Existing tools to record interaction data have several limitations [VCN+11, YR11, SRG15]. Mylyn data, for example, has a very coarse granularity making it hard (or impossible) to reconstruct the sequence of interaction events as they happened [YR11]. Our profiler, DFlow, records very fine-grained IDE interaction data, as they happen. Moreover, the data recorded by DFlow comply to a model for interaction data that we designed to provide structure to this novel source of information.

Most of the existing tools dealing with interaction data have a single, very specific, purpose. Navtracks, for example, keeps track of particular IDE interactions (*i.e.,* navigations) to support source code navigation [SES05]. Our DFlow, instead, is a general-purpose and extensible observer of developer interactions with the IDE. To demonstrate the versatility of our profiler,

we built a *recorder* that persists the recorded interaction data and sends it to our web server and the PLAGUE DOCTOR [MML15c], a tool similar to AUTUMN LEAVES [RND09] that aims at reducing the visual entropy inside the IDE. Moreover, developers can add their own extensions on top of DFLOW to study or support different aspects of the development process.

The next Part of this dissertation discusses our contribution in recording, modeling, and interpreting IDE interaction data. Chapter 5 sets the ground by presenting DFLOW, our non-intrusive interaction profiler for the *Pharo* IDE.

Part II

Modeling, Recording, and Interpreting Interaction Data

5

DFlow: Our Interaction Profiler for the Pharo IDE

This chapter discusses our experience and the contributions of our research in recording interaction data inside the IDE. All the interactions between developers and IDEs have an ephemeral nature, thus the first step to enable our research is to persist them. To do so, however, this raw data needs to be modeled. In this chapter we discuss a model for interaction data that we devised to provide structure to this heterogeneous source of information. We consider different kinds of events: From events that involve fine-grained interactions with the UI of the IDE to interactions that capture the mechanics of source code navigation. To record them we built DFlow, an interaction data profiler for the *Pharo* IDE. In this chapter we describe DFlow and how it evolved during our research.

Structure of the Chapter

Section 5.1 introduces DFlow, the interaction profiler for the *Pharo* IDE we developed to support our research. Section 5.2 describes the model that DFlow uses to structure interaction data. Finally, Section 5.3 describes the evolution of DFlow.

5.1 DFlow in a Nutshell

IDEs largely neglect interaction data, thus preventing further use of their potential [KM05]. To this aim we developed DFlow [MML15b], a shorthand for *"Development Flow"*. DFlow is a non-intrusive interaction profiler for the *Pharo* IDE.

Figure 5.1 schematizes the functioning of the most recent version of DFlow.

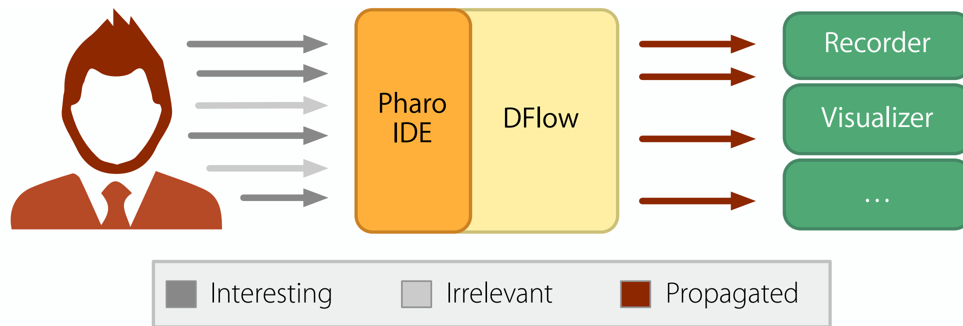


Figure 5.1. DFlow: Observing, filtering, and propagating IDE interactions

DFlow is an extension to the *Pharo* IDE to capture all the interactions of developers, structure them, and make them available for further use, *e.g.,* visualizing them. DFlow acts as a *filter* by ignoring interactions that we consider not relevant, *e.g.,* events triggered internally by the IDE to respond to the actual user interaction.

DFlow automatically collects data and periodically sends it to our server to support further analyses. As proof of concept, we developed approaches to support the development process on top of DFlow, discussed later in this dissertation (see Chapters 12 and 13).

The profiler collects more than 30 types of events (listed in Table 5.1), organized as per our model, described Section 5.2. For each event, it also records a timestamp down to millisecond precision. In the last years we distributed DFlow to different developers and collected millions of events.

5.2 A Model for Interaction Data

To provide a unified structure to interaction events, we devised a model, depicted in Figure 5.2. We identified three main categories of events according to their level of abstraction: meta events, user input events, and user interface events. Table 5.1 lists all the interaction events recorded by DFlow grouped by category.

5.2.1 Meta Events

Meta events (see Table 5.1.a) are all the interactions of the developer with program entities, *e.g.,* classes and methods. According to the *impact* of the interaction on the source code, we distinguish three categories of events: navigation, inspect, and edit.

Navigation events correspond to the events the developer performs while exploring source code entities, *e.g.,* selecting a method or a class in the code browser, opening a new code browser on a program entity, or performing a search with the dedicated UI.

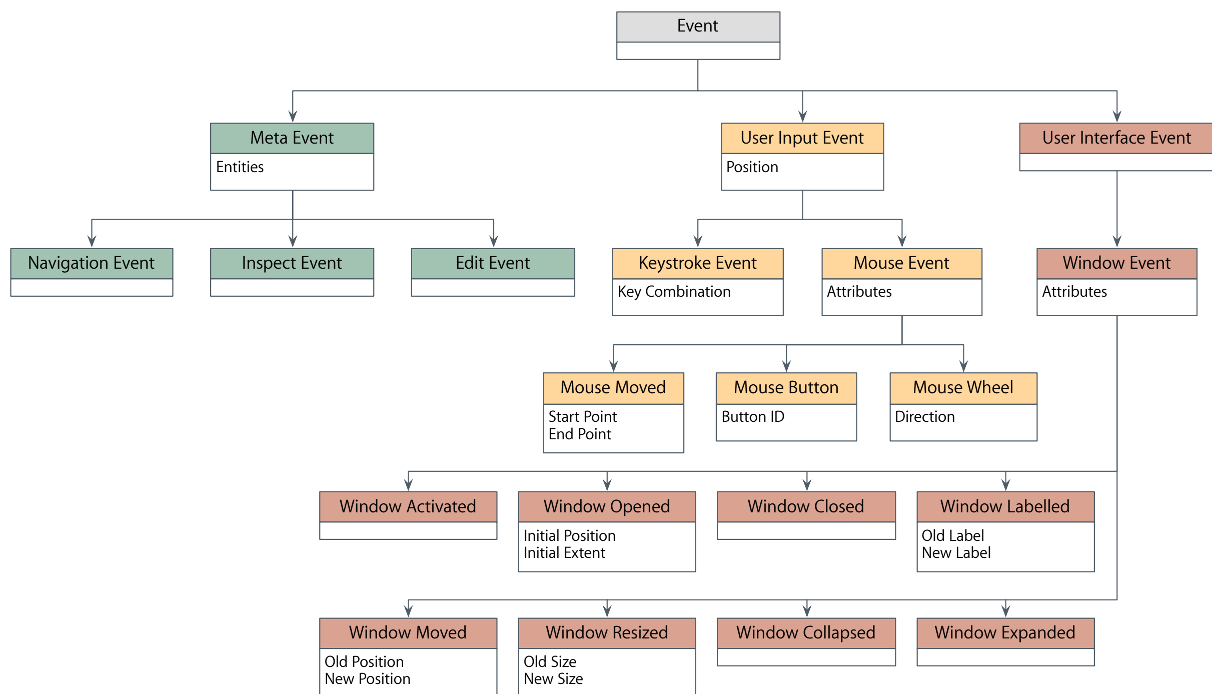


Figure 5.2. The model for interaction data of DFlow

Inspection events represent actions that a developer performs to understand the execution of a program. *e.g.,* debugging a piece of code or observing the value of a local variable or field (*i.e., watch expressions* in *Eclipse*).

Edit events represent actual source code modifications, *e.g.,* adding a new class or editing the body of a method. In most of the cases, meta events have one or more program entities associated to them, *i.e.,* when the developer modifies the source code of the method `Foo`, the corresponding DFlow meta event encodes this information for further analyses.

5.2.2 User Input Events

These are the events the developer performs using an input device, *e.g.,* mouse and keyboard. We distinguish two categories of events: mouse and keystroke events. All user input events recorded by DFlow (see Table 5.1.b) encode the position of the cursor when the event happens and other attributes. Keystroke events, for example, encode the current *key combination* that might be a single keystroke (*e.g.,* the key `'a'`) or a keyboard shortcut (*e.g.,* ⌘ `+'V'`).

Mouse events have different types with their own ad-hoc attributes. For example, the event that represents a *mouse move* encodes the initial and final position of the cursor before and after the movement. Other types of mouse events are mouse button events (*i.e.,* clicks) and interactions with the mouse wheel.

5.2.3 User Interface Events

These events represent the interactions of the developer with the user interface of the IDE (see Table 5.1.c). In the case of *Pharo*, the target IDE for our study, the user interface mostly consists of *windows*. Thus, user interface events are *window events*, such as opening, closing, or resizing

a window. Each event has its own attributes: A resize, for example, encodes the size of the window before and after the event itself.

Table 5.1. List of interaction data events recorded by DFlow

a) Meta Events	
NE_1	Opening a Finder UI
$NE_{2,3,4}$	Selecting a package, method, or class in the code browser
$NE_{5,6}$	Opening a system browser on a method or a class
NE_7	Selecting a method in the Finder UI
NE_8	Starting a search in the Finder UI
IE_1	Inspecting an object
IE_2	Browsing a compiled method
$IE_{3,4}$	Do-it/Print-it on a piece of code (*e.g.,* workspace)
$IE_{5,6,7}$	Stepping into/Stepping Over/Proceeding in a debugger
IE_8	Run to selection in a debugger
$IE_{9,10}$	Entering/exiting from an active debugger
$IE_{11,12}$	Browsing full stack/stack trace in a debugger
$IE_{13,14,15}$	Browsing hierarchy, implementors, or senders of a class
IE_{16}	Browsing the version control system
IE_{17}	Browse versions of a method
$EE_{1,2}$	Creating/removing a class
$EE_{3,4}$	Adding/removing instance variables from a class
$EE_{5,6}$	Adding/removing a method from a class
EE_7	Automatically creating accessors for a class

b) User Input Events	
$ME_{1,2}$	Mouse button up/down
$ME_{3,4}$	Scroll wheel up/down
ME_5	Mouse move
$ME_{6,7}$	Mouse-out/in
KE_1	Keystroke pressed

c) User Interface Events	
$WE_{1,2}$	Opening/closing a window
WE_3	Activating a window, *i.e.,* window in focus
$WE_{4,5,6,7}$	Resizing/moving/minimize/maximize a window

5.3 Evolution of DFlow

DFlow is a software system and, as such, it evolved during our research. This section details its evolution, from a manual tool to a fully automated approach.

5.3.1 A Manual Interface to Record Development Sessions

The first version of DFlow, offered a minimalistic UI, depicted in Figure 5.3, that developers use to start, pause, resume, and stop the recording of a session.

With this version of DFlow, when a developer is ready to record a new development session, she presses the *"Start"* button (Fig. 5.3-A) and the profiler starts to collect her IDE interactions. If the development is interrupted, say by a Skype call, the developer could press the *"Pause"* button (Fig. 5.3-B) to explicitly indicate this interruption. Then, at a later time, she could

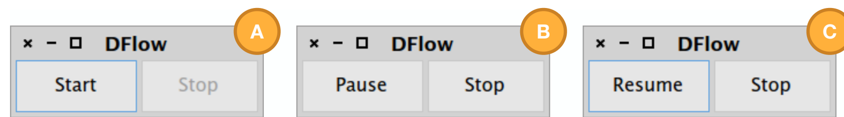


Figure 5.3. The UI of the manual version of DFlow

resume the session with the appropriate button (*i.e.*, *"Resume"*, see Fig. 5.3-C). At any moment, the developer can signal the end of the session with the *"Stop"* button (Fig. 5.3-A, B, and C).

At the end of the session, DFLOW asks for additional information, such as a brief description that explains the main purpose of the session and its *type*. The session type can be one of the following: *general purpose*, *refactoring*, *enhancement*, or *bug-fixing*. After recording this information, DFLOW stores the development session to our server.

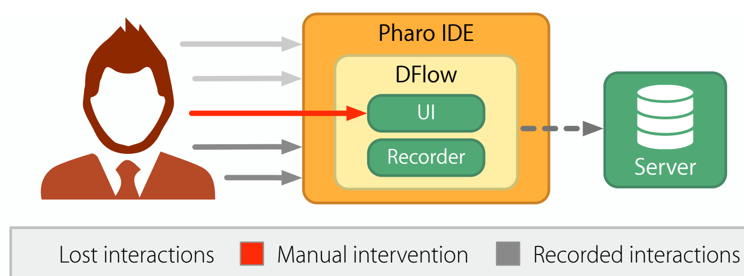


Figure 5.4. The functioning of the manual version of DFlow

Limitations. Figure 5.4 summarizes the functioning of this version of DFLOW that strongly relies on manual intervention of the developer. Often times developers forget to start a session or ignore the fact that they could pause a session to explicitly indicate the interruptions in their workflow. As a result, we miss some of their interaction data or we receive sessions without explicitly marked interruptions. Since we learned that *"we can not ask developers to push a button"* [Joh01], we decided to make DFLOW less disruptive, as discussed in the next section.

5.3.2 Automatic Recording of Development Sessions

To reduce manual user intervention, we implemented a fully automatic recording mechanism of interaction data, summarized in Figure 5.5.

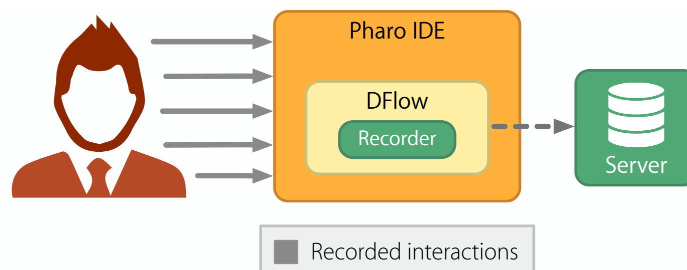


Figure 5.5. The functioning of the automatic version of DFlow

The new version of DFLOW no longer features the UI described in the previous section. Once a developer installs DFLOW, the profiler starts immediately to collect data and periodically sends it to our server. This is the version we used for the majority of our data collection process.

5.3.3 DF2low: Automatically Observing, Filtering, and Propagating Interactions

The most recent re-engineering of DFlow, summarized in Figure 5.6, decouples the profiling logic from the rest of the functionalities offered by DFlow, *e.g.,* recorder and visualizer.

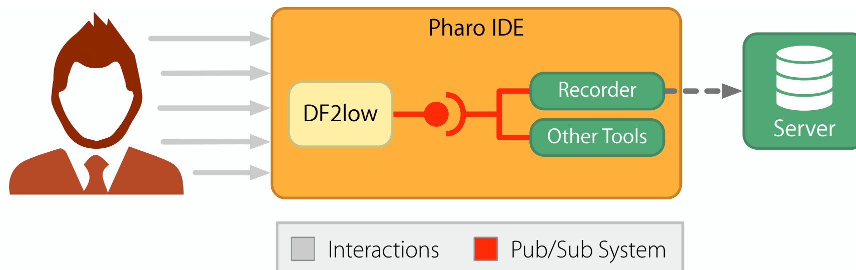


Figure 5.6. The functioning of DF2low, the most recent version of DFlow

In this version, also known as DF2low,[1] the profiler observes IDE interactions, filters out those that are not interesting, and makes the interesting development interactions available for further use. We illustrated this process also in Figure 5.1, at the beginning of this section.

Under the hood, DF2low uses *Announcements* [PHA11]: A framework for event notification originally designed by Vassili Bykov, former lead software engineer at *Cincom*.[2] The framework implements the Observer pattern [GHJV95] in a fully object-oriented fashion. Each event is an object that can be customized with data specific to the application. In this architecture, DF2low acts as a publisher (*i.e.,* or *Announcer*) of events. All other tools that are interested in using interaction data (*e.g.,* the recorder) can subscribe and register an action in response the events of the announcer. Our interaction data recorder, for example, forwards the events to the server to store the data for retrospective analyses.

In addition to *Announcements*, DF2low also replaces the original mechanism to profile *meta events* (see Section 5.2). In the previous versions of DFlow, we use a source code instrumentation framework called *Spy* [BBRR12] inspired by the original version of method wrapper [BFJR98]. This mechanism enables to inject code snippets code before or after the execution of a method. When we first started to implement DFlow, this seemed the best solution to achieve our goal. Unfortunately, for our purposes, *Spy* has a number of limitations. In the first place, since it is not part of the standard *Pharo* distribution, it takes time to be installed. In addition, in *Spy* all the instrumented methods[3] are replaced by template methods that invoke the injected code. For example, a method with two arguments is replaced by the following template:

```
with2arg: v1 arg: v2
  ^#metaObject run: #selector with: {v1.v2} in: self
```

Even though the source code of the vast majority of methods instrumented by DFlow is not interesting for most developers, the obfuscation performed by *Spy* may potentially hinder program comprehension tasks. These reasons encouraged us to find a better alternative to *Spy*.

DF2low, in fact, replaces *Spy* with *Reflectivity*. *Reflectivity* is a novel framework that realizes the concept of sub-method structural and behavioral reflection. Partial behavioral re-

[1] In the remainder of this document we will use the general term "DFlow" to indicate our interaction data profiler, independently of its version and and the underlying technologies used.

[2] See `https://www.cincom.com`

[3] Each meta event recorded by DFlow (see Table 5.1) corresponds to a method in the source code of the *Pharo* IDE, *e.g.,* the event IE_1 corresponds to the method "`inspect`" of the class `Object`.

flection was first introduced by Tanter *et al.* [TNCC03] that realized it at instruction level (*i.e.*, bytecode). Few years later, Denker revisited this concept and applied it on top of sub-method reflection [Den08, ch. 5]. *Reflectivity* is currently included in the *Pharo* distribution, thus reducing significantly the installation time of DFlow. In a nutshell, *Reflectivity* enables to *annotate* any node in the abstract syntax tree (AST) of *Pharo*, including method declaration nodes. These annotations, called *MetaLink* in *Reflectivity* jargon, are objects associated with an AST node that can alter the behavior of a running application.[4] *MetaLinks* have access to the *parameters* of the annotated node. For example, in a method declaration node, a link can access its arguments, receiver, and signature. In addition, the user has control on *when* to activate the link: before, after, or instead of the annotated AST node. Finally, a link can also have a dynamic activation condition. This determines if it needs to be activated or not depending on the current context. Figure 5.7 illustrates this concept: Behavioral reflection realized with sub-method reflection.

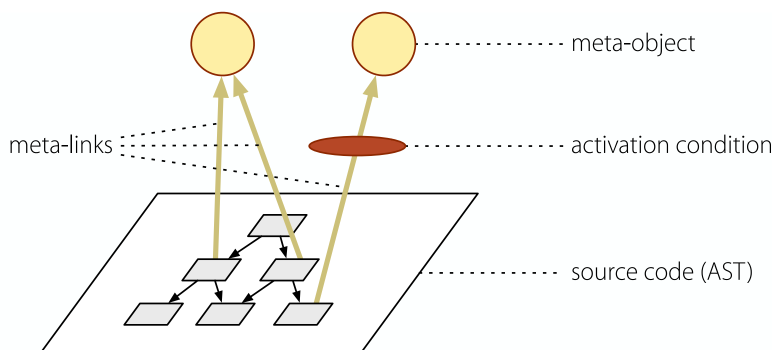


Figure 5.7. Partial behavioral reflection realized with sub-method reflection

To annotate an AST node, a developer creates a new *MetaLink* that describes the new behavior and attaches it to the AST node. An *annotation* (or *MetaLink*) is a snippet of code that will be executed whenever the annotated AST node is executed. For example, if we annotate the AST node corresponding to the method `Foo»#bar`[5] with a *MetaLink* that invokes the method `Halt»#now`[6] whenever the message `Foo»#bar` is executed, the annotation will be triggered and the system will hang, due to the breakpoint introduced by the annotation.

The user can specify the desired *control* (*i.e.*, execute the annotation before, after, or instead of the annotated AST node) and an activation condition, *i.e.*, activate annotations only if a particular condition is satisfied. A link specifies a *meta-object*. This is the object to which the annotation code will be executed. It can be any object, included the current AST node. In the breakpoint scenario, the *meta-object* would be the `Halt` class that to which the MetaLink will send the message `now`.

The following listing shows a concrete usage of *Reflectivity* to implement statement coverage [PY07], a well known testing technique.

```
"Creates a new MetaLink"
link := MetaLink new
            metaObject: #node;
            selector: #tagExecuted.
```

[4]Conceptually, this mechanism is similar to Aspect Oriented Programming (AOP), a technique to augment the behavior of a program in a non intrusive fashion [KLM+97].

[5]This *Smalltalk* notation refers to the method `bar` of the class `Foo`.

[6]In *Smalltalk*, this method triggers a breakpoint and stops the execution.

```
"Attaches the link to all the AST nodes of the message exampleMethod in the ReflectivityExamples class"
(ReflectivityExamples>>#exampleMethod) ast
   nodesDo: [:node | node link: link].
```

The example above creates a new *MetaLink* where the *meta-object* is the AST node itself (*i.e.,* `#node`). The selector is the code to be executed, in this example is the method `tagExecuted`. This link is then attached to all the AST nodes of the `exampleMethod` that belongs to the `ReflectivityExamples` class. This example usage implements the statement coverage testing technique: Every time a node is executed, it is *tagged* by sending it the `tagExecuted` message.

The key advantage of *Reflectivity* over *Spy* is altering the behavior of a program without changing the actual source code. All the annotations of *Reflectivity* are invisible to the end user and the source code of the annotated methods is left unchanged.

In the context of DFlow, we mainly use *Reflectivity* to profile meta events without changing the source code of the *Pharo* IDE. For example, to observe when a developer *inspects* an object we install the following MetaLink on the AST method declaration node of `Object»#inspect`:

```
link := MetaLink new
            metaObject: DF2ReflectivityProfiler;
            control: #before;
            selector: #'eventForReceiver:andSelector:withArguments:';
            arguments: #(#receiver #selector #arguments).
```

When the user *inspects* an object the MetaLink is activated and sends the message `eventForReceiver:andSelector:withArguments:` to the class `DF2Reflectivity- Profiler`. The *control*, sets to `#before`, specifies that the MetaLink should be activated called before the actual annotated node, *i.e.,* right before performing the real inspection of the object. The notation

```
arguments: #(#receiver #selector #arguments).
```

means that the MetaLink passes as parameters to the invoked method on the MetaObject the *receiver*, *selector*, and *arguments* of the actual method call. In the case of `Object»#inspect`, the *receiver* is the actual inspected object, the *selector* is `#inspect`, and *arguments* is an empty list, since `Object»#inspect` is a unary message, *i.e.,* does not take arguments. The *Reflectivity Profiler* is responsible of creating, and propagating, a new interaction event, *i.e.,* a DFlow Inspect Meta-Event, in this case.

5.4 Reflections

This chapter introduced our model that provides structure to the heterogeneous world of interaction data. Our model groups events in three categories: Meta, User Input, and User Interface events. Meta events are further categorized according to the impact they have on source code entities. To this aim, we distinguish between navigation, inspection, and editing events. Besides the model, our second contribution is DFlow, a general-purpose IDE interaction profiler for the *Pharo* IDE. In this chapter we explain DFlow, its high-level architecture and how it evolved over time, a meta event.

Modeling and recording interaction data are the first steps to enable our research. The following chapters describe approaches to interpret and make sense of the recorded interaction data. We propose approaches to estimate the role of program comprehension (Chapter 6), to reconstruct high-level development activities from fine-grained IDE interactions (Chapter 7), and to measure the navigation efficiency of developers in the *Pharo* IDE (Chapter 8).

6

A Naïve Model to Interpret Interaction Data

The previous chapter introduced our meta-model for interaction data and DFlow, our profiler for IDE interactions. Interaction data captures the behavior of software developers inside the IDE. However, in their raw form, interaction histories are only dense streams of events that are hard to understand. In this chapter we present a naïve model to interpret IDE interaction histories and estimate the time spent by developers in different activities. To gather insights from different contexts, we investigate two datasets, one recorded with DFlow and the other recorded with Plog, an interaction profiler for the *Eclipse* IDE [KKA12]. Our model estimates how much time developers spend to navigate, write, and understand source code. Among all software engineering activities, program understanding has been estimated to be one of the most challenging tasks performed by developers [LVD06]. According to Corbi, developers understand programs in different ways, for example by reading documentation, reading source code, and executing the program itself [Cor89]. During maintenance and evolution activities developers spend more time reading than writing source code [VMV95]. Another essential activity for program comprehension is navigating between code fragments [KM05, PFS+11]. Ko *et al.* estimated that developers spend 35% of their time navigating the system at hand [KMCA06]. Some studies claim that understanding absorbs about half of the time of developers [ZSG79, FH83, Cor89]. However, these facts have been taken for granted for quite some time, and some research fields, such as program comprehension and reverse engineering, base their reason to exist on such facts. This chapter investigates whether these facts can be confirmed.

Structure of the Chapter

In Section 6.1 we introduce the dataset of this study and the two recording tools that we employed. Section 6.2 introduces our naïve model to empirically measure the effort developers spend on program comprehension. In Section 6.3 we present and analyze our findings. Finally, in Section 6.4 discusses the limitation of our naïve approach which calls for a refinement of our estimation models, later described in Chapter 7.

6.1 Datasets and Recording Tools

This section gives a brief overview of the two recording tools we used (DFlow and Plog) and discusses how the data differs depending on the development context.

6.1.1 Interaction Events and Sessions Meta-Information

In this study we consider a specific kind of interaction data that we called "*meta events*".[1] We classify the events according to the impact they have on the involved program entities, as follows:

- *Navigation events*, used to browse (but not modify) source code entities, *e.g.,* opening a browser to list the methods of a class or a file to depict its contents;

- *Inspect events* (*Smalltalk*-only), that happen when developers examine the state of objects, for example during debugging;

- *Edit events*, that modify code, *e.g.,* adding a new class or modifying a method.

In addition to the *type*, each event has other properties associated to it: a *creation time* (the timestamp when it occurred) and a set of *program entities involved*, such as classes and methods. Sequences of interaction events compose what we call "*development session*".

With DFlow we collected 175 development sessions totaling more than 110,000 interaction events coming from 7 developers (both industrials and academics) "*in the wild*" [ABSN13]. With Plog, instead, we collected 15 sessions from 15 master students, totaling almost 4,000 events, during a controlled experiment part of the evaluation of Plog itself [KKA12].

At the time of this study DFlow was at its early stages and required manual intervention of the developer (see Section 5.3.1). Developers have to explicitly start the recording of the development session. In this occasion, we request developers to specify i) a *title* to briefly describe the aim of the session and ii) a *type* describing its intended purpose.[2] Developers can choose between 4 types: General purpose, refactoring, enhancement, and bug-fixing. Each development session has a *start* and an *end* timestamp, and an identifier that represents the *author* of the session. A session might last for hours or days, but the developer will probably not program uninterruptedly. We introduce the concept of *sub-sessions* to indicate pauses during development. In DFlow, when a developer stops programming for any reason (*e.g.,* a conference call) she can explicitly pause (and later resume) the recording. In addition, when we post-process the interaction histories of DFlow and Plog, we automatically detect (and remove) what we call "*idle times*" longer than 10 minutes and create implicit sub-sessions without these idle periods.

Since we use two different tools that track interactions in IDEs which support a different development philosophy, the information above cannot be mapped to the interaction data collected by the tools. To make an example: The *Pharo* IDE is a multi-window environment, and it is normal for users to spawn several windows during development. Moreover, it is an IDE based on program entities, and not files, which entails that a developer is looking always at methods in isolation. As opposed to that, *Eclipse* is an IDE based on tabs and files, which requires that developers open files which are presented in the editor and it is therefore normal that a developer has several methods in front of his eyes in the same window. This in essence means that in the *Eclipse* case it is not possible for us to unambiguously understand which entity is being looked at. However, in this context we are not interested into *what* exactly the developer is doing, but

[1] As introduced in Section 5.2, meta events are all the interactions that involve program entities.
[2] This holds only for *Smalltalk* sessions recorded with DFlow.

more *when* she is doing it. To get a feeling for the type of data we are considering, refer to Figure 6.1, which shows a *Smalltalk* development session at a glance.

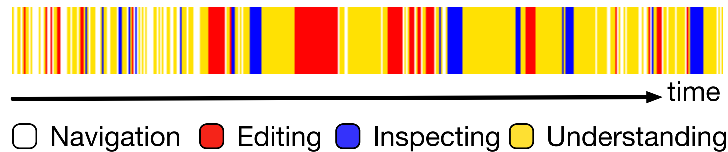


Figure 6.1. A development session at a glance

This session lasted 30 minutes and 43 seconds. During that time a developer triggered a total of 455 interaction events out of which 403 navigations (*i.e.,* white), 21 inspections (*i.e.,* blue), and 31 edits (*i.e.,* red). Navigations are very quick "trigger" events, such as opening a browser, clicking on class or method names to see their contents, *etc.* Inspections happen when the developer opened a special UI (called *inspector*, in *Smalltalk* dialect) to examine the state of an object at run-time. Finally, edit events represent the actual "writing" of the source code by editing new or existing code. The *rest*, depicted in yellow, is the time when the developer was seemingly "doing nothing", but in fact this is the time when the developer sits in front of the source code and looks at it. It thus represents the actual program understanding part. In the example of Figure 6.1, navigation events are mostly present at the beginning of the session, when the developer is getting his bearings in the system. Moreover, editing events are nearly always present after a longer understanding time interval.

6.1.2 DFlow and Smalltalk Interaction Histories

Table 6.1 and Table 6.2 summarize the DFlow dataset, grouped per type and per developer. It counts 175 sessions by 7 developers, totaling more than 110,000 events.

Table 6.1. Dataset – Smalltalk sessions data per type

	Sessions			Events						Windows	
				Navigation		Inspect		Edit			
Type	#	Avg. #Subs.	Avg. Duration [hh:mm:ss]	#	Avg.	#	Avg.	#	Avg.	#	Avg.
BUG	27	2.11	44:38	7,154	264.96	1,205	44.63	1,537	56.93	1,739	64.41
ENH	86	2.26	57:08	40,973	476.43	3,631	42.22	3,992	46.42	7,331	85.24
GEN	55	4.18	1:33:38	42,831	778.75	1,767	32.13	4,196	76.29	6,372	115.85
REF	7	1.71	48:46	3,294	470.57	22	3.14	331	47.29	224	32.00
All	**175**	**2.38**	**1:06:21**	**94,252**	**538.58**	**6,625**	**37.86**	**10,056**	**57.46**	**15,666**	**89.52**

BUG: Big-Fixing – ENH: Enhancement – GEN: General Purpose – REF: Refactoring

Developers are all from the *Pharo* open-source community, with a background in both industry and academia, located in 4 different sites (INRIA Lille, France; University of Bern, Switzerland; University of Santiago, Chile; University of Lugano, Switzerland). The vast majority of the sessions are marked Enhancement and General, where General sessions are usually the longest, with an average length of more than 1.5 hours. We also note that bug-fixing sessions are generally short, which can be explained by the fact that often they were dedicated and guided sessions to fix particular known bugs in existing code. Across all session types, navigation events are one order of magnitude more than the number of editing events. General purpose sessions are, on average, those that contain more sub-sessions and have the higher number of navigation

and edit events (778.75 and 76.29 on average). Bug-fixing sessions have the highest number of inspect (44.63 on average). This is not surprising since inspects are often triggered while debugging. It appears that, while fixing bugs, developers navigate less than in other sessions (264.96 navigations on average). Our hypothesis on this is that these sessions are highly focused since the developer already knows the subset of program entities involved in a given bug-fixing task. One third of times developers do not know what they are going to do, *i.e.,* they select "general purpose" as session type. Those sessions have a very high concentration of navigation and edit events (respectively 778.75 and 76.29) and a higher number of windows with respect to other types (115.85 where the average over all the sessions is 89.52). This justifies their general, broad purpose: Developers do a little bit of everything.

Table 6.2. Dataset – Smalltalk sessions data per developer

Dev.	Sessions			Events						Windows	
	#	Avg. #Subs.	Avg. Duration [hh:mm:ss]	Navigation #	Navigation Avg.	Inspect #	Inspect Avg.	Edit #	Edit Avg.	#	Avg.
SD1	12	6.08	03:01:24	21,617	1,801.42	183	15.25	2,458	204.83	3,144	262.00
SD2	3	1.00	16:27	393	131.00	157	52.33	24	8.00	71	23.67
SD3	65	1.49	52:32	20,468	314.89	2,157	33.18	2,091	32.17	3,183	48.97
SD4	6	1.83	48:13	2,183	363.83	353	58.83	1,196	199.33	608	101.33
SD5	70	2.84	56:26	35,495	507.07	2,952	42.17	3,289	46.99	7,336	104.80
SD6	7	4.29	01:25:18	6,862	980.29	337	48.14	472	67.43	555	79.29
SD7	12	6.67	01:34:25	7,234	602.83	486	40.50	526	43.83	769	64.08
All	**175**	**2.82**	**01:06:21**	**94,252**	**538.58**	**6,625**	**37.86**	**10,056**	**57.46**	**15,666**	**89.52**

Table 6.2 gives insights on how different developers behave. All developers, but SD4, have a number of navigation events that are one order of magnitude more than the number of editing. It remains to be investigated the behavior of SD4 that on average performs one edit event every two navigations. Developer SD1 has a very high number of windows per session (262.00). She is developing a visualization engine for *Smalltalk*. *Pharo* is a window-based environment and her tool generates visualization inside windows. This explains why her use of windows is significantly higher than others. This developer, in general, is an outlier: She has significantly higher number of navigations and edits with respect to other developer and further investigation on her behavior is required. Developers SD3 and SD5 are the two subjects that used DFlow the most. The former is pretty new to the *Smalltalk* programming language. She navigates and edits less than the average of other developers. SD5, an experienced *Smalltalk* developer, instead is mostly in line with average values. It remains to be investigated how the expertise of developers impact on their behavior inside the IDE. Developers SD1 and SD6 are the two subjects that navigate the most (1,801.42 and 980.29 events on average). Interestingly, while SD1 uses a very high number of windows, SD6 despite navigating a lot more than other developers, uses few windows. Developers SD1 and SD7 have the highest numbers of sub-sessions (6.08 and 6.67, where the average is 2.82), symptoms of highly interrupted sessions.

6.1.3 Plog and Java Interaction Histories

In addition to *Smalltalk* interaction histories collected with DFlow, we also analyzed 15 *Java* development sessions captured using the Plog tool developed by Kobayashi *et al.* [KKA12].

Plog captures interactions at two granularities: file and method level. For this study we only used histories at file level which are comparable to the interaction histories recorded with

DFlow (a class in *Smalltalk* is conceptually similar to a file in *Java*, which often contain one class). Interaction histories captured by Plog have the following meta-information: i) an *author name*; and ii) a list of *events*, with the following information: i) a *timestamp* when the event was recorded and ii) a *type*. Plog records two types of events:

- N_R: a pure navigation, *i.e.,* without any editing;
- N_W: a navigation with an editing event.

Navigation events happen when the developer moves between tabs and opens new tabs. N_R events are comparable to navigation events in DFlow, while N_W events contain implicit editing events. Inspection events are not captured in Plog, as *Eclipse* does not offer a live programming environment, as opposed to *Pharo*.

Table 6.3 summarizes the data collected with Plog. There are 15 sessions by 15 different developers. Naïvely, Plog has no concept of sub-session, but we pre-processed the data to automatically identify sub-sessions according to periods of idle (minimum idle set to 10 minutes) in the interaction histories. This generates 144 sub-sessions without idle periods.

Table 6.3. Dataset – Java sessions data per developer

Developer	# Subs.	Duration [hh:mm:ss]	Events	
			Navigation	Edit
JD1	5	1:15:39	48	17
JD2	7	2:02:51	121	13
JD3	11	3:49:00	208	56
JD4	5	1:28:38	139	48
JD5	14	4:42:31	248	82
JD6	8	2:48:00	204	57
JD7	18	12:38:03	803	295
JD8	5	1:28:24	111	11
JD9	6	2:18:31	121	11
JD10	19	3:36:59	231	61
JD11	7	1:48:24	109	20
JD12	11	2:01:47	121	21
JD13	1	1:28:37	70	28
JD14	16	5:16:42	454	27
JD15	11	2:35:52	132	29
All	**144**	**49:19:58**	**3,120**	**776**

Developers were given different tasks, namely to extend an existing system to fulfill a series of change requests without any prior knowledge of the system (*i.e.,* enhancement sessions). Each developer has a personal way of approaching the task, as some of them took several hours (up to 12) to implement the requested changes, while others took little bit more than one hour. We can infer that about 20% of all navigation events led to an editing activity.

6.2 Naïve Estimation Model

Our goal is to use interaction data to estimate how much time developers spend to navigate, write, and understand code. Figures 6.2 and 6.3 depict two development sessions captured with our recording tools.

Figure 6.2. Visualizing *Java* development activities

Figure 6.2 shows a *Java* session recorded with PLOG. That session lasted 1 hours 48 minutes and 24 seconds and counts 109 navigation activities (white), 20 edits (red), and a large amount of understanding (yellow). The developer (JD11 in Table 6.3) spends 65.1% of her time understanding the system whilst performing perfective maintenance. The first half of the session is essentially composed of understanding driven by the navigation of the system at hand. After that, the developer acquired the necessary knowledge to perform the changes. The second part of the session encloses all the 20 editing activities interleaved with navigation events and understanding time frames. Editing activities have different durations. The first one lasts for a quite long time, probably due to the fact that the developer is not confident with her understanding of the system. Afterwards, with the increased confidence in the system, she performs a series of short editing activities.

Figure 6.3. Visualizing *Smalltalk* development activities

Figure 6.3 shows a *Smalltalk* session recorded with DFLOW. It lasted for 1 hour 14 minutes and 8 seconds and counts 491 navigations, 25 inspections, and 34 edits. The first difference with Figure 6.2 is the presence of inspection activities (blue). In *Smalltalk* an inspection happens when a developer observes some property of an instance of an object, *i.e.,* the value of its fields. Inspections are, most of the times, triggered while debugging when a developer wants to investigate the reasons of a failure. Figure 6.3 depicts a bug-fixing session of the developer SD5 (in Table 6.2). The developer spent 60.91% of her time in understanding tasks, 15.28% on editing activities, and 10.93% navigating between code fragments. In the session, there are some peculiarities. For example, inspections have often an edit preceding them. Our hypothesis is that the developer first changes the code, then executes and debugs it. This is possible in the *Pharo Smalltalk* IDE as it is a live programming environment, *i.e.,* even when the system raises an exception it is still "alive" and can be modified on the fly. Most important editing activities (*i.e.,* the ones with the longest durations) are often preceded by significant understanding time frames. This is a symptom of the fact that developers want to gather a substantial understanding of the system prior to changing it. As in the *Java* session, editing activities are concentrated in the second part of the session while the first part is mainly comprehension.

6.2.1 Modeling DFlow Interaction Histories

Figure 6.4 depicts a fragment of a raw interaction history recorded with DFLOW, that is, a sequence of events with their timestamp. As a base to estimate the amount of program understanding during development, we need first to estimate the amount of time spent for other development activities, starting from the recorded events.

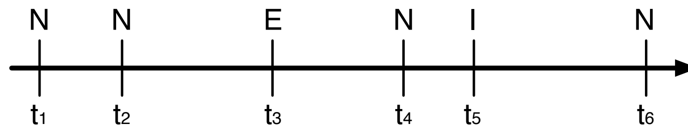


Figure 6.4. A raw interaction history recorded with DFlow

DFlow interactions contains three types of events: navigation, inspection, and editing, for which we estimate the duration of the corresponding activities as follows.

Navigation Activities

Navigation events are clicks in the user interface of the IDE. To perform the "click" a user spends a relatively small amount of time. In addition to the click, a navigation implies an additional time required to move to the target area for the click. This is known as Fitts's Law and computed as a function of the distance and the size of the target [Fit54]. We approximate this time to a fixed average duration (ΔN). Figure 6.5 shows the updated interaction history after estimating navigation activities.

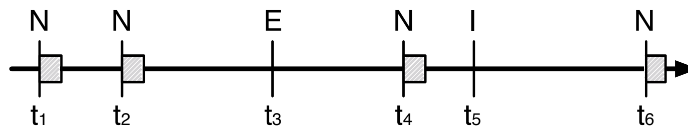


Figure 6.5. DFlow interaction history with Navigation Activities

Editing Activities

For an editing activity E_i, DFlow records an event when the user is done with the editing: We denote this time as $end(E_i)$. We assume that the duration of the edit (ΔE_i) is a fraction (P_E) of the time interval between the end time of the previous activity $end(prev(E_i))$ and the end time of E_i.

$$\Delta E_i = P_E \times (end(E_i) - end(prev(E_i))) \tag{6.1}$$

Figure 6.6 shows the interaction history after assigning a duration to editing events.

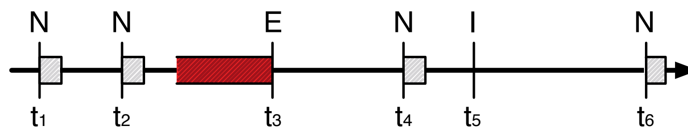


Figure 6.6. DFlow interaction history with Navigation and Editing Activities

Inspection Activities

For an inspection activity I_i, DFlow records an event when the user starts inspecting, *i.e.,* $start(I_i)$. The duration of the inspection (ΔI_i) is a fraction (P_I) of the time interval between $start(I_i)$ and the start time of the following event $start(next(I_i))$.

$$\Delta I_i = P_I \times (start(next(I_i)) - start(I_i)) \quad (6.2)$$

Figure 6.7 shows the situation after assigning a duration to inspection events.

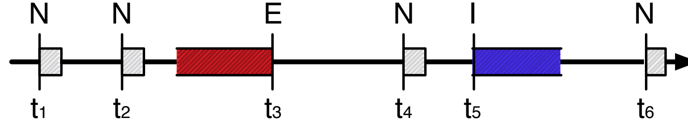


Figure 6.7. DFlow interaction history with Navigation, Editing, and Inspecting Activities

Understanding Activities

Understanding activities are all the *gaps* between the other types of activities. Everything that is not navigation, inspection, and editing is program understanding. For this reason, we first assigned duration to the other types of events, and we identify every remaining gap in the interaction history as understanding activities.

Figure 6.8 shows the final interaction history, with all the activities.

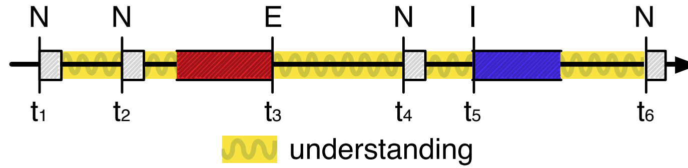


Figure 6.8. DFlow interaction history with all the Activities

Interaction between Inspection and Editing

Our estimation model uses time intervals between events to estimate the duration of the corresponding activities. When an inspection is followed by an editing, our model is more complex. The duration of the inspection activity depends on the duration of the editing activity, which in turn depends on the duration of the inspection.

Figure 6.9 illustrates the situation.

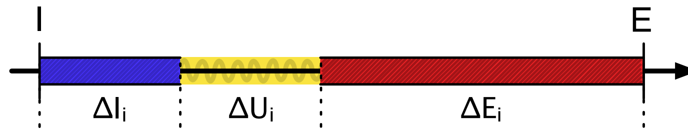


Figure 6.9. The case of editing after inspection

According to our estimation model, the duration of the inspection activity ΔI_i and the duration of the editing activity ΔE_i are as follows:

$$\Delta I_i = P_I \times (start(next(I_i)) - start(I_i)) \quad (6.3)$$

$$\Delta E_i = P_E \times (end(E_i) - end(prev(E_i))) \quad (6.4)$$

To solve Equation (6.3) we should know the start time of $next(I_i)$, that is E_i. We cannot know this before solving Equation (6.4).

In turn, to determine ΔE_i we need the end time of $prev(E_i)$, that is I_i, and we cannot know this *a priori.* We can rewrite Equations (6.3) and (6.4) as follows:

$$\begin{cases} \Delta I_i = P_I \times (\Delta - \Delta E_i) \\ \Delta E_i = P_E \times (\Delta - \Delta I_i) \end{cases} \tag{6.5}$$

where $\Delta = \Delta I_i + \Delta U_i + \Delta E_i$. ΔI_i depends on ΔE_i, and vice-versa. We want to find two percentages, P_I^{IE} and P_E^{IE}, such that ΔE_i and ΔI_i can be computed from the whole interval Δ:

$$\begin{cases} \Delta I_i = P_I^{IE} \times \Delta \\ \Delta E_i = P_E^{IE} \times \Delta \end{cases}$$

By definition, these two fractions are:

$$\begin{cases} P_I^{IE} = \frac{\Delta I_i}{\Delta} \\ P_E^{IE} = \frac{\Delta E_i}{\Delta} \end{cases}$$

By dividing Equations (6.5) by Δ we obtain:

$$\begin{cases} P_I^{IE} = P_I \times (1 - P_E^{IE}) \\ P_E^{IE} = P_E \times (1 - P_I^{IE}) \end{cases} \tag{6.6}$$

Solving Equations (6.6) we obtain:

$$\begin{cases} P_I^{IE} = \frac{P_I - P_I \times P_E}{1 - P_I \times P_E} \\ P_E^{IE} = \frac{P_E - P_I \times P_E}{1 - P_I \times P_E} \end{cases}$$

To summarize, the three time intervals depicted in Figure 6.9, can be computed in function of the whole interval Δ as follows:

$$\begin{cases} \Delta I_i = P_I^{IE} \times \Delta \\ \Delta E_i = P_E^{IE} \times \Delta \\ \Delta U_i = \Delta - \Delta I_i - \Delta E_i \end{cases}$$

Reflections

We do not have "perfect" interaction data. For example, during an editing activity we do not know exactly what the developer is doing, only that she is editing. Our future work in this context is to track interactions at finest level possible, *i.e.,* the keystroke level for editing operations, and mouse events (including scrolling, *etc.*) for navigation and inspection activities.

6.2.2 Modeling Plog Interaction Histories

PLOG interaction histories are composed of two types of events: N_R and N_W. The former represents pure navigation events, while the latter represents a navigation event that follows an editing activity. Figure 6.10 shows a PLOG interaction history.

Since PLOG interaction histories lack inspection events, and editing events are implicit, to estimate the duration of developer activities we use a slightly different model with respect to the case of DFLOW histories.

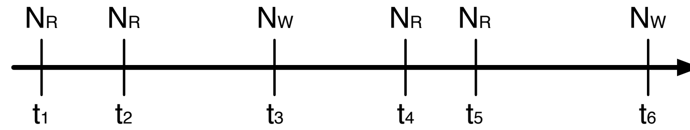


Figure 6.10. A raw interaction history recorded with Plog

Navigation Activities

Events of type N_R denote pure navigation. As in the DFLOW case, we approximate this time to a fixed duration (ΔN). Figure 6.11 shows the updated interaction history with explicit pure navigation activities of fixed duration.

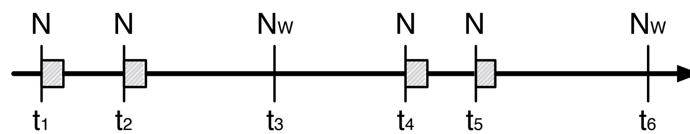


Figure 6.11. Plog interaction history with Navigation Activities

Editing Activities

In PLOG editing activities are implicit: The tool records an event of type N_W at time $t(N_W)$ when the user performs a navigation after editing. The actual editing happened in an unknown moment in the time interval between the end of the previous event, denoted as $end(prev(N_W))$ and $t(N_W)$. To make the editing activity E_i explicit, we place it in the middle of the interval with a duration (ΔE_i) that is half the duration of that interval. The N_W event is then converted to a navigation activity with a fixed duration, ΔN.

$$\Delta E_i = P_E \times (t(N_W) - end(prev(E_i))) \tag{6.7}$$

Figure 6.12 shows the updated interaction history assigning a duration to both navigation and editing activities.

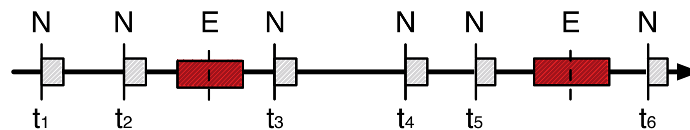


Figure 6.12. Plog interaction history with Navigation and Editing Activities

Understanding Activities

We assign every remaining gap in the interaction history to understanding activities. Figure 6.13 shows the final shape of the PLOG interaction history.

Reflections

The recorded interaction histories are not perfect. Since we do not have keystroke-level events, we must approximate the editing activities. Is it possible that if a user open a new file and

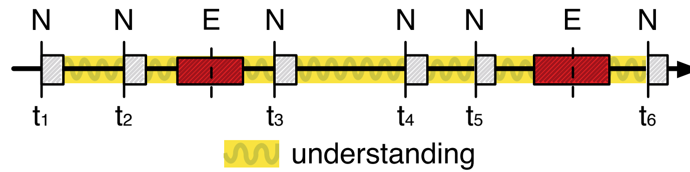


Figure 6.13. Plog interaction history with Navigation and Editing Activities

spends one minute in that file, 59 seconds could be spent on doing nothing. We believe our approximation is however reasonable, as it does not assign the editing activity an exaggerated weight. Future work in this context is the recording of keystroke-level events, as done for example in EclipseEye [Sha07].

6.2.3 Discussion: The Degrees of Freedom of the Models

The estimation model we propose to quantify development activities has three degrees of freedom: i) ΔN, that represents the conventional average duration of navigation events; ii) P_E, that models the average percentage of editing activities between an edit event and the preceding event; and iii) P_I, which similarly represents the percentage of inspection activities in Smalltalk interaction histories. We intend *navigation* as the "mechanics of navigation", *i.e.,* the clicks in the user interface. Thus, we conventionally assume that each navigation event lasts, on average, 0.5 seconds. We fix ΔN to 0.5s.

Quantifying the right amount of P_E and P_I is out of the scope of this work, since it would require more fine-grained events to be collected (*e.g.,* keystroke events). Instead, we discuss how the results of our quantification model change by varying these parameters, obtaining possible lower and upper bounds to the amount of time spent in program understanding. Table 6.4 shows how the amount of program understanding changes by varying P_E and P_I in the case of Smalltalk development sessions collected by DFlow.

Table 6.4. Results – Amount of Understanding in *Smalltalk* sessions varying P_E and P_I

P_I \ P_E	**0.10**	**0.25**	**0.50**	**0.75**	**0.90**
0.10	**88.44%**	83.74%	75.87%	67.94%	63.16%
0.25	86.17%	81.63%	73.97%	66.20%	61.46%
0.50	82.34%	78.02%	70.71%	63.24%	58.62%
0.75	78.46%	74.28%	67.28%	60.16%	55.73%
0.90	76.10%	71.97%	65.09%	58.16%	**53.93%**

The most pessimistic estimate is shown in the bottom right cell of the table, and corresponds to $P_E = P_I = 90\%$. In this case understanding amounts to 54% of time, which is slightly above the upper-bound of previous estimates [ZSG79, FH83, Cor89]. The most optimistic estimate is on the top-left cell of the table, corresponding to $P_E = P_I = 10\%$: In this case, understanding consumes around 88% of the time of developers, which is significantly above previous estimates. Obviously, more accurate estimates lay between these two quite unrealistic extremes, and they are shown in the other cells of the table. It appears very likely that actual time spent by developers in program understanding has been underestimated by previous research by at least 10-20%.

Table 6.5 shows similar results in the case of *Java* development sessions recorded by PLOG. In this case, since the tool did not record inspection events, the only parameter to vary is P_E. Pessimistic and optimistic estimates for program understanding are similar to the case of DFLOW, suggesting similar considerations about the time spent in program understanding.

Table 6.5. Results – Amount of Understanding in Java sessions varying the estimate of P_E

P_E	**0.10**	**0.25**	**0.50**	**0.75**	**0.90**
	94.33%	87.13%	75.12%	63.11%	**55.91%**

6.3 Results

Tables 6.6, 6.7, and 6.8 summarize the distribution of development activities for all the recorded sessions in the case of DFLOW and PLOG. For each development activity, we calculate the average of the percentage of time spent by developers in each activity.

As we pointed out in the previous section, program understanding is a dominant activity, as it accounts, on average, from 54 to 94% of the total development time in each session. The values are similar despite the profound difference in the way developers produce *Java* and Smalltalk code. This suggests that the role of program understanding has been underestimated by previous research. Decades ago researchers claimed that program understanding absorbs about 50% of the time of developers [ZSG79, FH83, Cor89], while we find that the percentage is likely to be higher.

Figure 6.14 depicts the distribution of the relative importance of the activities using box plots. Please note that inspection (INS) only applies to *Smalltalk* sessions recorded with DFLOW. Although there can be substantial differences between individual sessions and developers, what emerges is a clear pre-dominance of program understanding activities.

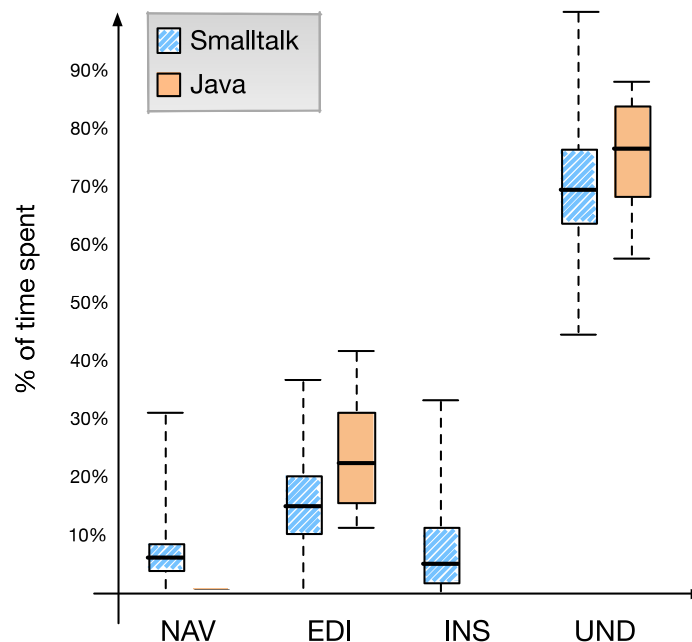


Figure 6.14. Development activities for all sessions

Role of Session Types

While in interaction histories collected by DFLOW and PLOG program understanding is dominant in both cases, there is difference ranging from 2% to 6% depending on the estimates of parameters, as shown in Table 6.6. On all the Smalltalk sessions, program understanding ranges from 54 to 88% while on *Java* sessions these bounds range from 56 to 94%. A possible interpretation of this difference can be found in the fact that PLOG sessions were all enhancement sessions, while the sessions we collected with DFLOW are of different type, *e.g.,* bug-fixing and refactoring sessions.

Table 6.6 shows the different distribution of the duration of development activities per session type in the case of Smalltalk sessions collected by DFLOW.

Table 6.6. Results – Development activities per session type

Type	N (%)	E (%)			I (%)			U (%)		
		Min	Med	Max	Min	Med	Max	Min	Med	Max
DFlow: Smalltalk										
ENH	7.8%	30.6%	17.4%	3.6%	13.4%	7.8%	1.7%	48.2%	67.0%	86.9%
REF	9.1%	30.4%	16.9%	3.4%	1.7%	0.9%	0.2%	58.9%	73.1%	87.3%
GEN	6.2%	21.3%	12.1%	2.5%	9.5%	5.5%	1.2%	63.0%	76.2%	90.1%
BUG	4.6%	25.5%	14.5%	3.0%	17.4%	10.0%	2.1%	52.6%	70.9%	90.3%
All	**6.8%**	**26.9%**	**15.3%**	**3.2%**	**12.3%**	**7.2%**	**1.6%**	**54.0%**	**70.7%**	**88.4%**

ENH: Enhancement – REF: Refactoring – GEN: General Purpose – BUG: Big-Fixing

Plog: Java										
All	**0.9%**	**43.2%**	**24.0%**	**4.8%**	–	–	–	**55.9%**	**75.1%**	**94.3%**

Table 6.7. Results – Smalltalk development activities per developer

Dev.	N (%)	E (%)			I (%)			U (%)		
		Min	Med	Max	Min	Med	Max	Min	Med	Max
SD1	5.1%	30.9%	17.6%	3.6%	13.7%	8.0%	1.7%	50.3%	69.4%	89.5%
SD2	8.9%	21.6%	12.1%	2.4%	20.3%	11.3%	2.3%	49.2%	67.7%	86.4%
SD3	8.0%	30.2%	17.0%	3.5%	5.6%	3.3%	0.7%	56.2%	71.8%	87.9%
SD4	8.5%	26.5%	15.1%	3.1%	12.8%	7.4%	1.6%	52.2%	69.0%	86.8%
SD5	4.6%	16.8%	9.7%	2.1%	9.3%	5.6%	1.3%	69.4%	80.1%	92.1%
SD6	8.4%	10.5%	6.0%	1.2%	10.5%	6.0%	1.2%	70.6%	79.6%	89.1%
SD7	5.7%	22.3%	12.6%	2.6%	10.5%	6.1%	1.3%	61.5%	75.6%	90.4%

The role of understanding is still dominant in all session types: However, there is some significant difference in the distribution of editing and navigation activities. Refactoring sessions, for example, have a high understanding component and make minimal use of inspection activities (0.2-1.7%). While program understanding does not appear to be significantly different between session types, there are some differences on the distribution of other activities. Enhancement and Bug Fixing sessions spend significant time on inspection (around 10%), while Enhancement and Refactoring sessions spend more time on editing.

Developer Diversity

Table 6.7 lists the relative importance of the activities for each Smalltalk developer. We can deduce diverse “profiles”: SD3 has a tendency towards more navigation and editing, and less

understanding. In essence he could be characterized as more "aggressive" towards the code base. Similarly, SD2 spends almost the same amount of time on navigation, but distributes almost equally the remaining time between editing and inspection, being thus more "cautious" and she probably frequently verifies the implemented changes. SD5 and SD6 are even more cautious, denoted by their high understanding values. In essence they reflect more on the code before they change it. SD3 however is making very little use of inspecting, which is a preferred activity of skilled developers. SD1 and SD3 have the highest editing and the among the lowest understanding values. In essence, they seem to be at ease with the code base and confidently change it without the need to rely on extensive navigation.

Table 6.8 lists the relative importance of the activities for each *Java* developer.

Table 6.8. Results – Java development activities per developer

		E (%)			U (%)		
Dev.	**N (%)**	Min	Med	Max	Min	Med	Max
JD1	0.5%	60.2%	33.4%	6.68%	39.3%	66.0%	92.78%
JD2	0.8%	20.3%	11.3%	2.25%	78.8%	87.9%	96.90%
JD3	0.8%	52.5%	29.1%	5.83%	46.8%	70.1%	93.41%
JD4	1.3%	32.0%	17.8%	3.56%	66.6%	80.9%	95.12%
JD5	0.7%	68.7%	38.2%	7.63%	30.6%	61.1%	91.64%
JD6	1.0%	40.7%	22.6%	4.52%	58.3%	76.4%	94.46%
JD7	0.9%	62.1%	34.5%	6.90%	37.0%	64.6%	92.22%
JD8	1.1%	24.9%	13.8%	2.76%	74.0%	85.1%	96.19%
JD9	0.7%	28.3%	15.7%	3.14%	71.0%	83.5%	96.13%
JD10	0.9%	34.1%	18.9%	3.79%	65.0%	80.2%	95.32%
JD11	0.9%	27.6%	15.4%	3.07%	71.5%	83.8%	96.05%
JD12	0.8%	49.1%	27.3%	5.46%	50.0%	71.9%	93.71%
JD13	0.7%	75.2%	41.8%	8.35%	24.2%	57.6%	90.98%
JD14	1.2%	22.4%	12.4%	2.48%	76.4%	86.4%	96.32%
JD15	0.7%	50.1%	27.9%	5.57%	49.1%	71.4%	93.72%

From this data we can infer developer "profiles". For example JD8 took a short time to implement the changes, but has high understanding and low editing (refer to Table 6.3 for durations). JD2 and JD14 have a similar behavior but they took more time to implement the task. We can say that JD8 is someone who thinks deeply about what to do, and then does it quickly and firm. JD13 is at the opposite end of the spectrum: The high amount of editing time, with relatively high navigation time, denotes a developer who heavily meanders in the code base before she slowly implements the changes.

6.3.1 Threats to Validity

Construct Validity. We presented a model to estimate the duration of development activities starting from recorded interaction histories. A threat to validity for our results is the accuracy of this model, that may not precisely capture, for example, the moment when editing activities start. To get a more accurate model to estimate activity durations, one should record more fine-grained interaction data and corresponding events. However, in Section 6.2.3 we described how even varying the degrees of freedoms the essence of this work remains true.

Moreover, our study only considered the part of software development carried on inside an IDE. Since program comprehension can be carried out throughout the whole software development lifecycle, our findings are a lower bound of the total time devoted to software understanding.

Statistical Conclusion. We considered a total of 190 sessions with around 120,000 interactions, which we consider to be substantial enough to deduce some conclusions. However, we did not measure the statistical confidence of our results.

External Validity. The weight of different development activities may significantly vary with different languages and IDEs. To mitigate this possible threat, we considered two significantly different programming languages and IDEs, obtaining similar estimates that give us confidence about the generalizability of our results.

A similar argument can be formulated about the developer diversity, which may influence the amount of time required from program comprehension. In our study, we considered 15 different *Java* developers in the case of interaction data recorded on the *Eclipse* IDE, and 7 different developers with different background and experience in the case of Smalltalk and the *Pharo* IDE. Further investigation is needed to understand how developers' expertise influences the way they interact with the UI of the IDE.

6.4 Reflections

Raw interaction histories are dense streams of events that require to be interpreted to draw insightful conclusions. In this chapter we discussed a naïve model to estimate the time spent by developers in various development activities. We collected interaction data from 22 developers, 15 working in Java with *Eclipse* and 7 working in *Smalltalk* with *Pharo*, totaling hundreds of hours of recorded activities. Among the goals of our study we wanted to gather empirical evidence about the role of program comprehension. Researchers claimed the program understanding is one of the most time consuming activities of software development. This claim is taken as an *ipse dixit*, a dogmatic statement to be accepted as it is. Our preliminary results show that in the last 40 years the role of program comprehension may have changed and that this topic needs further investigation, also to motivate the importance of research areas such as program comprehension, reverse engineering, and mining software repositories.

Our naïve models have some limitations, discussed in Section 6.3.1. To mitigate them, we devised a more precise model to infer high-level development activities from low-level IDE interaction histories, detailed in the next chapter.

7

Inferring High-Level Development Activities from Interaction Histories

The previous chapter introduced a naïve model that only uses meta event to estimate the role of program understanding in development sessions. This chapter extends the previous model by also considering other kinds of interaction data to provide precise estimates of high-level development activities such as code editing and program understanding. While being a fundamental part of software development, it is unclear how program comprehension is supported by modern IDEs. They offer various tools and facilities to support the development process, like i) Code Editors, ii) Code Browsers, and iii) Debuggers [GLD05, GGD07], but none of these components is dedicated to program comprehension. Instead, comprehension emerges from the complex interleaving of such activities. Moreover, researchers discovered that some UI paradigms (*e.g.,* windows- or tabs-based IDEs) may negatively influence development, hindering comprehension and generally developer productivity [RND09]. While this claim is intuitively convincing, there is no quantitative evidence on how much time is spent on fiddling with the UI of an IDE.

In this chapter we propose an inference model of development activities to measure the time spent in editing, navigating and searching for artifacts, interacting with the UI of the IDE, and performing activities such as inspection and debugging. We applied our model on a dataset of 740 development sessions coming from 18 developers totaling about 200 hours of development time and more than 5 million events. Our results show that program comprehension absorbs more time than generally assumed, and that fiddling with the UI of IDEs can substantially hinder the productivity of developers.

Structure of the Chapter

Section 7.1 summarizes the dataset and highlights the different types of interaction data considered by this study. Section 7.2 describes the inference model and Section 7.3 presents an in-depth analysis of how developers spend their time. Finally, Section 7.4 concludes the chapter by reflecting upon the implication of this study and playing the devil's advocate.

7.1 The Dataset

This sections summarizes the dataset for this study and details the interaction events used to devise a more precise estimation model for development activities.

7.1.1 More Than Meta Events

Differently from the naïve model presented in Chapter 6, in addition to meta events, the new model also considers other types of interaction data. In particular, as per the model for interaction data presented in Section 5.2, DFlow also records *User Input* and *User Interface* that we also call *Low-level events*. *Low Level Events* are events that deal with input devices (*e.g.,* mouse, keyboard), or the user interfaces of the IDE. In particular, DFlow records:

- *Window events*[1] are all the events that deal with the different windows of the *Pharo* IDE, like opening, closing, moving or resizing a window;
- *Mouse events* include movements, scrolls, and mouse clicks inside the UI of the IDE. Each event captures the cursor position.; in particular, movement events are specialized when the mouse moves outside the main *Pharo* window to other areas of the screen (*mouse-out* event) or back inside (*mouse-in* event);
- *Keyboard events* represent the keystrokes happening in the session. Each event records the keystroke (or combination of keystrokes with modifiers like *command* or *shift*) that has been typed.

7.1.2 Facts and Figures

Our dataset, summarized in Tables 7.1 and 7.2, is composed of 738 development sessions totaling 197 hours of development and 5 million events. The first two columns report anonymized identifiers of the developer with their total number of sessions collected with DFlow. A "session" is a sequence of IDE interactions without periods of inactivity that last more than 5 minutes.[2] We call these periods *"idle periods"* (or *"idle time"*). DFlow detects when the developer is away from the keyboard and splits the interactions into multiple sessions (discarding the idle period). For each developer we collected: i) the recording time, ii) the number of low-level events, iii) the number of meta events, and iv) the number of windows used during development sessions. Each row in the table reports values for a single developer. The leftmost part of the table reports the total values, while the rightmost part the average values (per session). In the last row (*i.e.,* All) total values accumulate the values for all developers while average values are computed using a weighted arithmetic mean across all developers weighted on the number of sessions.

To recruit participants we sent a call on the Pharo-Dev mailing list[3]. Eighteen developers, both professionals and academics, answered the call and helped us in the collection of their interactions. Participants were not assigned specific tasks. Instead they have been working on their own personal projects, *a.k.a. "in the wild"* [ABSN13].

They all share a common code base (*i.e.,* the open source code of *Pharo*) but we have no information on the size of their own private projects. The dataset features 2 Master students, 9 Ph.D. students, and 7 professionals. We distinguish 3 levels of expertise, *i.e.,* how many years they have been programming in *Pharo*. D9 is the only developer that can be considered a *novice*.

[1] *Pharo* is a window-based IDE, thus UI events are interactions with windows, *i.e.,* window events.
[2] This value was chosen arbitrarily.
[3] See `http://pharo.org/community`

Table 7.1. Dataset – Total values grouped by developer

Dev.	#S	Total Rec. Time	Total Low Level ME	KE	WE	All	Total #Meta	Total #Win
D1	407	89h 21m 46s	1,436,332	104,622	16,402	1,557,356	80,030	3,966
D2	136	52h 09m 52s	1,945,028	143,852	33,801	2,122,681	58,468	5,677
D3	76	28h 50m 44s	596,928	66,717	9,376	673,021	35,168	2,080
D4	32	06h 17m 34s	129,492	6,441	1,426	137,359	6,653	539
D5	19	01h 31m 45s	21,575	1,709	344	23,628	1,087	90
D6	14	04h 20m 26s	62,857	8,628	449	71,934	4,037	132
D7	11	04h 13m 54s	82,294	9,670	1,573	93,537	3,201	453
D8	9	01h 14m 47s	19,550	328	103	19,981	441	34
D9	9	03h 19m 54s	26,970	3,194	732	30,896	2,120	231
D10	8	01h 03m 33s	14,797	1,232	252	16,281	1,471	67
D11	5	01h 12m 38s	33,775	2,521	283	36,579	3,510	80
D12	5	01h 07m 48s	31,186	2,554	321	34,061	1,381	89
D13 ⋆	2	00h 05m 56s	3,332	273	54	3,659	12	10
D14	1	01h 52m 38s	10,420	551	920	11,891	5,033	182
D15 ⋆	1	00h 01m 58s	714	21	11	746	32	5
D16	1	00h 15m 57s	4,741	565	60	5,366	305	20
D17 ⋆	1	00h 04m 55s	1,347	49	33	1,429	3,423	8
D18 ⋆	1	00h 07m 46s	5,197	38	82	5,317	292	28
All	**738**	**197h 13m 54s**	**4,426,535**	**352,965**	**66,222**	**4,845,722**	**206,664**	**13,691**

Table 7.2. Dataset – Average values grouped by developer

Dev.	#S	Avg. Rec. Time	Avg. Low Level ME	KE	WE	All	Avg. #Meta	Avg. #Win
D1	407	00h 13m 10s	3,529.07	257.06	40.30	3,826.43	196.63	9.74
D2	136	00h 23m 01s	14,301.68	1,057.74	248.54	15,607.95	429.91	41.74
D3	76	00h 22m 46s	7,854.32	877.86	123.37	8,855.54	462.74	27.37
D4	32	00h 11m 48s	4,046.63	201.28	44.56	4,292.47	207.91	16.84
D5	19	00h 04m 50s	1,135.53	89.95	18.11	1,243.58	57.21	4.74
D6	14	00h 18m 36s	4,489.79	616.29	32.07	5,138.14	288.36	9.43
D7	11	00h 23m 05s	7,481.27	879.09	143.00	8,503.36	291.00	41.18
D8	9	00h 08m 19s	2,172.22	36.44	11.44	2,220.11	49.00	3.78
D9	9	00h 22m 13s	2,996.67	354.89	81.33	3,432.89	235.56	25.67
D10	8	00h 07m 57s	1,849.63	154.00	31.50	2,035.13	183.88	8.38
D11	5	00h 14m 32s	6,755.00	504.20	56.60	7,315.80	702.00	16.00
D12	5	00h 13m 34s	6,237.20	510.80	64.20	6,812.20	276.20	17.80
D13 ⋆	2	00h 02m 58s	1,666.00	136.50	27.00	1,829.50	6.00	5.00
D14	1	01h 52m 38s	10,420.00	551.00	920.00	11,891.00	5,033.00	182.00
D15 ⋆	1	00h 01m 58s	714.00	21.00	11.00	746.00	32.00	5.00
D16	1	00h 15m 57s	4,741.00	565.00	60.00	5,366.00	305.00	20.00
D17 ⋆	1	00h 04m 55s	1,347.00	49.00	33.00	1,429.00	3,423.00	8.00
D18 ⋆	1	00h 07m 46s	5,197.00	38.00	82.00	5,317.00	292.00	28.00
All	**738**	**00h 16m 02s**	**5,998.01**	**478.27**	**89.73**	**6,566.02**	**280.03**	**18.55**

The others are quite familiar with the *Pharo* IDE, with an expertise between 1 and 5 years (6 developers) or longer than 5 years (11 developers). Table 7.3 shows demographics information.

Table 7.3. Dataset – Demographics of developers

Developer	**Role**	**Expertise (Years)**
D9	Master Student	< 1
D14	Master Student	1–5
D1, 2, 10, 15, 16	PhD Student	1–5
D3, 12, 13, 18	PhD Student	> 5
D4, 5, 6, 7, 8, 11, 17	Professional	> 5

Our dataset features 738 sessions amounting to 197 hours of *actual* development time, *i.e.,* in the table, the total (and average) recording time column do not include the timespans in which the developers were idle (*i.e.,* DFLOW recorded no interactions with the IDE for more than 5 minutes). The dataset includes more than 5 million of events (*i.e.,* both meta and low-level). Sessions, on average, last for 16 minutes and count ca. 7,000 events. DFLOW recorded events for more than 13,000 windows, an average of 18.55 per session.

The total number of low-level mouse and keyboard events (*i.e.,* mouse and keyboard) is significantly (and not surprisingly) larger with respect to meta events, which begs the question whether these low-level events are related to meta-events. For example, sequences of mouse events can be related to specific entity inspections or navigation, but also with simple UI fiddling or adjustment.

There are substantial differences between different developers. The first 8 developers' average session time varies from 4 minutes and 50 seconds to almost 23 minutes and 5 seconds. This pinpoints the differences in their programming flow: Since recording time is free of idle time, this value is the "pure" time the developers spent in doing actual work. A developer with a short session time is a developer whose development flow is highly fragmented. Among the first eight developers, D2 and D7 are the developers with the less fragmented flow: Their sessions last, on average, more than 23 minutes, a duration which is in line with, for example, time management methods such as the *"Pomodoro Technique"* of extreme programming developed by Cirillo [Cir09].

On the other hand, developers like D5 and D8 have a fragmented flow: They work on average for around 4m 50s and 8m 19s respectively before having an interruption of at least 5 minutes. This corroborates the findings of LaToza *et al.* who established that developers are frequently interrupted, and that recovering from the interruptions can be difficult [LVD06].

Observing the distribution of low-level and meta events per developer we can speculate on how developers use the IDE. For example, on average D2 triggers more low-level events with respect to other developers (on average 15,607.95, more than twice the overall average). An interpretation for this is that she is constantly fiddling with the UI of the IDE to better accomodate her needs. Researchers already pointed out possible problems in dealing with the UIs of IDEs. For example, Rötlisberger *et al.* called *window plague* the problem developers might have while dealing with multiple windows or tabs [RND09]. Developer D8, instead, seems to be at ease with the UI of the IDE, since her number of low-level events is well below average and, in particular, she has the lowest average number of window events per session.

The number of meta events can be a rough indicator of productivity: They represent actions like creating/removing a class/method, or exploring code artifacts or inspecting objects. In terms of meta events, D2, D3, and D7 seem to be the most productive developers, while D5 and D8 are the less productive ones. This correlates with the fact that the development flow of D5 and D8 is more fragmented than the one of D2 and D3.

7.2 Inferring High-Level Development Activities

This section details our inference model for development activity and illustrates how we decompose software development into different categories.

7.2.1 Events, Sprees, and Activities

Interaction histories are streams of interaction. Each event has a timestamp, but it has virtually no duration, *e.g.,* DFLOW records the moment in time when a keystroke happens, but it does not have information about for how long the user pressed the key. However, this is not a problem, as our goal is to group these interaction events into sequences of higher level events for which it is easy to measure the precise duration.

Our model uses the concept of *reaction time* to aggregate events. While typing a piece of code, for example, a developer performs a sequence of keystrokes. These keys are separated by small pauses, in the range of milliseconds, due to the physical actions required involved, *i.e.,* pressing keys on a keyboard. In this time, the developer is focused on the writing activity per se. When the sequence of keystrokes terminates, the developers pause, reflecting on the just written piece of code, and planning the next steps. The *reaction time* is the time that elapses between the end of a physical action sequence (typing, moving the mouse, *etc.*) and the beginning of concrete mental processes like reflecting, planning, *etc.* which represent the basic moments of program understanding.

We denote the reaction time with RT and assign the duration of 1 second to it. This duration, known also as "Psychological Refractory Period" [Pin99] varies among humans, also depending on the task at hand, between 0.15 and 1.5 seconds. This might appear as a threat to validity, but as iterating through all possible values in that range did not affect our findings, we settled on the 1 second compromise, which is a conservative choice.

Our inference model uses the reaction time to group low-level events into higher-level abstractions: *mouse (or keyboard) sprees* and *development activities.*

Mouse/Keyboard Sprees

A spree is a sequence of mouse/keyboard events where each subsequent pair of events satisfy the following temporal and conceptual constraints:

- The time elapsed between them is smaller than the reaction time RT.
- They are performed on the same *window* of the IDE.
- Between them there is no *trigger event*, *i.e.,* a *meta event* which conceptually breaks a spree. Examples include adding or editing a method, navigating in the code browser, or inspecting an object. Please refer to Figure 3.1 for the explanation of the UIs of *Pharo*.
- At most one of its events should conceptually initiate a new spree or terminate the current one, *e.g.,* the keyboard shortcut `<Shift-cr>` that triggers a search action in the *Pharo* IDE initiates a new spree while the mouse moving outside the IDE terminates the current spree.

Development Activities

An activity is a sequence of mouse/keyboard sprees satisfying a number of constraints. We identify three kinds of activities:

- *Search Activities* are all the activities where the user performs a search inside the IDE (*e.g.,* on the Finder UI, Figure 3.1.5).
- *Inspection Activities*: examining an object by means of an inspector (see Figure 3.1.3).
- *Browser Activities* are all remaining activities after removing both search and inspection activities. They happen on specialized windows of *Pharo*, like the code browser, such as editing and navigation.

7.2.2 Inference Model in Practice

Figure 7.1 exemplifies our two-step process to construct development activities from raw interaction histories. The timeline on top shows a sample recorded interaction history, *i.e.,* a sequence of low-level and meta events.

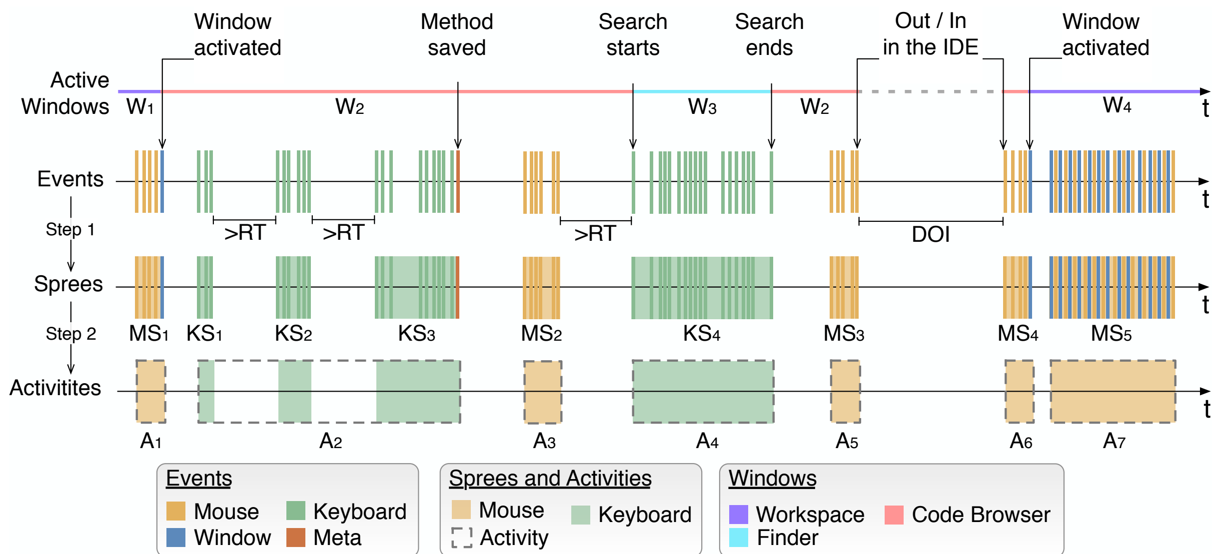


Figure 7.1. Sprees and Activities from fine-grained interaction histories

Step 1: *From Events to Sprees*

The first step towards the construction of activities is aggregating the events into mouse and keyboard sprees. At the beginning of the sample interaction history shown in Figure 7.1 there is a sequence of mouse events. We construct a new mouse spree (MS_1) by adding these events until one of the interrupting conditions is met. In this case, the reaction time is not elapsed (the difference between the last mouse event and the following event is smaller than RT), but there is a window event that activates a new window. MS_1 is complete. The next event in the sequence is a keystroke. We start composing a new keyboard spree (KS_1). After adding some keystrokes to it, the reaction time elapses (the difference between the timestamp of the next event and the last event in the current spree is greater than RT), thus we finalize KS_1. The same situation happens for both KS_2 and KS_3. In the case of KS_3, however, there is also a meta event of type EE_5 (see Table 5.1), *i.e.,* the action a developer performs to either add or edit the method of a class. We call this a **trigger** event that we associate to the current spree, *i.e.,* KS_3. MS_2, the next mouse spree, is interrupted due to the expiration of the reaction time. KS_4 is a keyboard spree that

starts when the user invokes the action that triggers the search in the *Pharo* IDE. Its stopping condition is the end of the search. The next mouse spree, MS_3 is interrupted because the mouse moves outside the *Pharo* IDE window. The time between the end of the spree is marked as DOI (Duration Outside IDE). The next event, a mouse event in this case, originates the next mouse spree, MS_4, interrupted due to the change of the window in focus. The last mouse spree, instead, is a dense sequence of mouse events with interleaving window events (not window activations, as they would have triggered the end of the spree). The timeline in the middle shows the results of this step: From dozens of low-level events we generated 5 mouse sprees and 4 keyboard sprees.

Step 2: *From Sprees to Activities*

The second step is to aggregate the sprees into high-level development activities. From the refined interaction history with sprees (*i.e.,* the middle timeline in Figure 7.1) our approach extracts, in sequence, search, inspection, and browsing activities. A spree can be part of a single activity, thus when we assign sprees to activities we *mark* them as already used. A search activity can be either performed on a Finder UI or triggered by a keyboard shortcut to start/confirm/abort a "spotlight-like" search (*i.e.,* <Shift-cr> to start the search, <cr> to confirm it, or either a mouse click or the <esc> keystroke to abort it). In this case, there is a search activity composed of the single key spree KS_4, triggered by the spree containing the shortcut <Shift-cr>. Inspect activities are performed on an inspector or triggered by inspection meta events (see Table 5.1). In the sample interaction history there are neither inspection meta events nor inspector windows, thus there are no inspection activities. All the remaining sprees are aggregated into "browser activities". Starting from the beginning of the interaction history, MS_1 is the first activity. The activity is interrupted because the next spree is on a different window due to the window activation at the end of MS_1. The next three keyboard sprees happen on the same window, and thus they get grouped into a single activity. The following activity is composed by the single mouse spree MS_2, because the next spree, KS_4, is marked as part of another activity, A_4. MS_3, the next mouse spree, creates an activity because there is an interruption, *i.e.,* out of the IDE. The second to last activity is only composed of the spree MS_4, because then there is a window focus change. Finally, the last remaining spree, MS_5, concludes the interaction history and makes up the last activity. The bottom timeline in Figure 7.1, shows the final result: From 9 sprees we end up with 1 search activity and 6 browser activities.

Our dataset is thus reduced from 5 million of recorded events to 174,366 sprees and to 31,609 development activities.

7.2.3 Decomposing Software Development

We decompose development time into the following distinct and disjunct categories: *understanding*, *navigation*, *editing*, and *UI interactions*.

Understanding (U)

Understanding (or program comprehension) time, aggregates three main components: (i) Basic Understanding, (ii) Inspection, and (iii) Mouse Drifting.

1. The *Basic Understanding (BU)* is the sum of all the basic moments of program understanding. It is represented by all time intervals between sprees which are longer than the *reaction time*. Basic understanding can be performed inside development activities (*i.e.,* intra-activities) and across subsequent activities (*i.e.,* inter-activities).

- BU_{intra} is the *Basic Intra-Activity Understanding Time* that is the sum of all the time intervals, longer than RT, between the sprees composing an activity.
- BU_{inter} is the *Basic Inter-Activity Understanding Time* that is the sum of all the time intervals, longer than RT, between subsequent activities.

2. *Inspection (I)* is the time a developer spends in inspection activities (mostly using inspector windows), computed as the sum of the duration of all the sprees that have as trigger an inspection meta event (see Table 5.1).

3. *Mouse Drifting (MD)* is the time the user "drifts" with the mouse without clicking. It is computed as the sum of the duration of the mouse sprees that are only composed or mouse movements, and no clicks. We also recorded the screen casts of several of the sessions collected by DFLOW and discovered that a large part of this time is absorbed by what we call *mouse-supported reading*, *i.e.,* when a developer uses the mouse as a "pointer" to support the reading of source code (*e.g.,* MS_2 in Figure 7.1).

Navigation (N)

Navigation time is the time spent in browsing through software [SES05]. This time includes both navigation using code browsers or package explorers and searching for particular program entities or pieces of code.

1. *Browsing (B)* is the time the developer spend while navigating between program entities. It is computed as the sum of the duration of the sprees that have as trigger a navigation meta event (see Table 5.1).

2. *Searching (S)* is the time a user spends in searching particular program entities such as methods or classes. This can be achieved using UIs such as the Finder UI (see Figure 3.1.5) or dedicated keyboard shortcuts, *e.g.,* `<Shift-cr>` in the *Pharo* IDE triggers a search dialog, see Figure 3.1.6. This time is the sum of the duration of the sprees happening inside user interfaces that support search activities. We remove from this time both the *user interactions*, *mouse drifting*, and *editing* time that might happen inside search UIs.

Editing (E)

Editing time is the time that the developer spend editing source code. This is computed by summing up the duration of all the sprees that have as trigger an editing meta event (see Table 5.1). For browsing activities, this definition is refined depending on the window where the activity is performed. In a code browser, for example, all the keystroke sprees that have no trigger for navigation contribute to editing time. Examples are KS_1, KS_2, and KS_3 in Figure 7.1.

User Interface Interaction (UI)

UI time is the time explicitly devoted in fiddling with the UI. This includes, for example, moving or resizing windows to better organize the IDE. It is computed as the sum of the duration of the mouse sprees that have interleaving window resize and move events in their timespan. An example is MS_5 in Figure 7.1.

Time Spent Outside the IDE (OI)

Besides the three categories discussed before, we also track the time spent outside of the IDE, *i.e.,* the time that the developer spend outside the *Pharo* IDE window. It is computed by summing up all the timespans that elapse between all activities that terminate with the *Pharo* IDE losing focus (*e.g.,* mouse goes outside the main IDE window) and the beginning of the next activity in the interaction history. It is denoted as DOI (*i.e.,* Duration Outside the IDE) in Figure 7.1.

7.3 How Developers Spend Their Time

Figure 7.2 summarizes how developers spend their time, on average, on each of the five high-level activities identified from fine-grained IDE interaction histories.

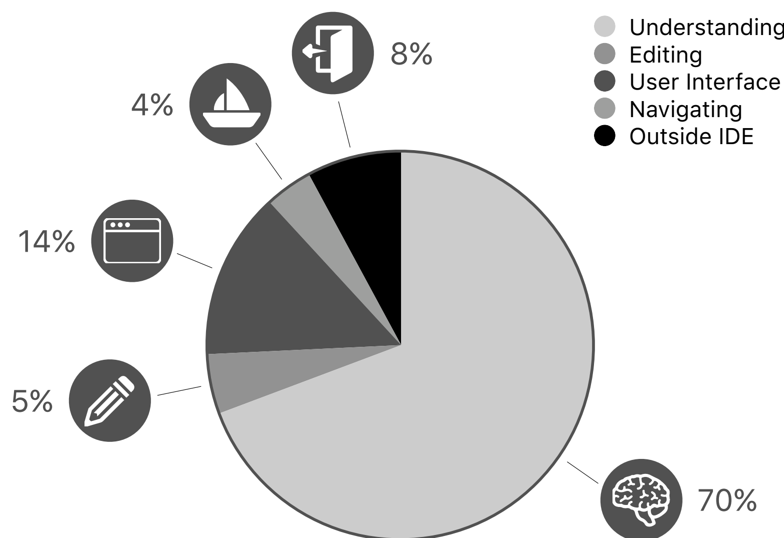


Figure 7.2. How do developers spend their time?

Program understanding is as expected the dominant activity, but as we see our analysis attributes to it even more importance than what the common knowledge suggests, reaching a value of roughly 70%. This is a strong point in favor of the research done in the field of program comprehension and reverse engineering. A rather worrisome finding is the time spent in UI interactions: roughly 17% of the time is spent in fiddling with the user interface of the IDE. The relatively small amount of time spent in editing and navigation (roughly 5% for both of them) is not surprising. In the case of editing it corroborates previous research, which established that when it comes to actual writing of source code the so-called "productivity" of developers is rather low [KM05]. This is yet another argument against measuring productivity with metrics like lines of code. In the case of navigation it emphasizes the fact that programming is not only writing, but rather a complex mental activity where a software system is perceived and navigated like a *graph* composed of nodes (*i.e.,* program entities) and edges (*i.e.,* relationships and dependencies between them), and not like a flat collection of textual artifacts. Last, the time spent outside of the IDE, *during* a session, corroborates the findings of LaToza *et al.* [LVD06]: Developers are often exposed to micro-interruptions of voluntary nature (*e.g.,* emails, instant messages, social networks notifications). Our dataset establishes that roughly 10% of the development time is spent on such interruptions. Table 7.4 presents the detailed results aggregated per developer.

The remainder of this section details the components of program understanding and the impact of work fragmentation on UI and understanding time.

Table 7.4. Results – Time components aggregated per developer

	Understanding (%)										
	Basic		Other			**Navigation (%)**			**Edit**	**UI**	**OI**
Dev.	**BU$_{intra}$**	**BU$_{inter}$**	**I**	**MD**	**Tot.**	**B**	**S**	**Tot.**	**(%)**	**(%)**	**(%)**
D1	35.07%	36.37%	0.25%	2.96%	74.64%	2.37%	0.38%	2.75%	3.07%	9.01%	10.53%
D2	37.41%	6.65%	3.23%	5.51%	52.79%	4.81%	1.19%	6.00%	9.76%	28.51%	2.94%
D3	47.68%	22.22%	0.87%	3.76%	74.54%	4.47%	0.26%	4.73%	5.44%	12.21%	3.08%
D4	38.06%	27.86%	0.53%	3.28%	69.74%	3.21%	0.14%	3.35%	3.75%	14.13%	9.03%
D5	22.90%	45.67%	0.07%	1.89%	70.53%	1.20%	0.00%	1.20%	2.74%	10.90%	14.63%
D6	52.85%	23.40%	0.11%	2.25%	78.61%	3.41%	0.05%	3.46%	9.18%	8.76%	0.00%
D7	56.77%	10.70%	0.07%	2.06%	69.59%	1.82%	0.00%	1.82%	10.67%	17.57%	0.35%
D8	45.66%	24.84%	0.00%	3.60%	74.09%	2.26%	0.00%	2.26%	1.29%	11.34%	11.03%
D9	58.68%	17.93%	0.73%	0.93%	78.26%	1.00%	0.09%	1.09%	6.08%	13.45%	1.12%
D10	36.94%	28.34%	0.57%	1.96%	67.81%	5.41%	0.00%	5.41%	4.10%	22.41%	0.28%
D11	39.11%	7.77%	0.00%	4.27%	51.14%	5.90%	0.00%	5.90%	6.66%	11.59%	24.70%
D12	28.58%	8.25%	0.00%	4.24%	41.07%	2.51%	0.00%	2.51%	10.49%	31.73%	14.20%
D13 ⋆	52.97%	15.36%	0.30%	4.67%	73.29%	1.47%	0.00%	1.47%	4.07%	21.16%	0.00%
D14	7.04%	86.76%	0.07%	0.21%	94.08%	0.56%	0.00%	0.56%	1.24%	4.13%	0.00%
D15 ⋆	54.14%	22.58%	0.00%	1.54%	78.26%	1.73%	0.00%	1.73%	2.66%	17.34%	0.00%
D16	64.80%	1.09%	2.93%	3.47%	72.28%	1.39%	0.16%	1.54%	10.01%	15.91%	0.26%
D17 ⋆	73.74%	3.48%	0.00%	0.85%	78.07%	4.46%	0.00%	4.46%	1.97%	15.50%	0.00%
D18 ⋆	29.15%	6.71%	0.00%	4.21%	40.06%	4.16%	0.00%	4.16%	1.32%	33.45%	21.01%
Avg.	**37.82%**	**27.70%**	**0.87%**	**3.46%**	**69.85%**	**3.09%**	**0.47%**	**3.56%**	**4.90%**	**13.81%**	**7.88%**

7.3.1 The Components of Program Understanding

The attentive reader has probably noted that Figure 7.2 does not include some of the components described in the previous section, such as inspection and mouse drifting – even if their contribution is relatively low, it is not negligible. They have not been elided, instead, we grouped them as components of program understanding.

Inspection is an activity, performed on objects at runtime, to check their status, and ultimately to understand the dynamic aspects (*i.e.,* the behavior) of the code. It is essential in any process involving running code, like debugging. In live environments like *Pharo*, inspection can be used to inspect any runtime object created by running any piece of code, *i.e.,* it directly supports the understanding of run-time behavior.

Mouse drifting is another component of program understanding that corresponds to mouse movements without any apparent consequent action. One of the typical examples of mouse drifting is to support the reading of a piece of code: Developers support the reading activity by slowly moving the mouse pointer as a guide to read and understand the code.

There is large variability of program understanding among developers. In our dataset, it ranges from 41% for D12 to 94% for D14; however, in both cases, we do not have many recorded sessions, so they are probably simply outliers for specific tasks that require respectively a minimal or a maximal amount of understanding. In case of D12, most of the remaining time is actually spent on fiddling with the UI (around 32%) and being outside the IDE (ca. 14%), which suggests she is not concentrated on the task at hand.

Inspection also varies between developers: For example D2 spends around 3% of her time in inspection activities, while the average inspection time is below 1%. Similar higher time spent on inspection can be seen on D16. Higher variety is present on the usage of mouse drifting. D2 and D12 spend much of their time fiddling with the UI.

Our data provides insights on how understanding is distributed among activities. On average, basic inter-activity understanding amounts to 10% more than intra-activities understanding. It is evident that base understanding is prevalent inside activities, that is, inside conceptually related sequences of keyboard or mouse sprees. In other words, the process of program understanding is not really an activity per-se, but it is interleaved with other activities like editing. Again, there is significant variability between developers. The process of base understanding for D1, for example, is almost equally divided between intra- and inter-activity understanding. For D5 and D14, there is significantly more inter-activity understanding, which probably means that the activities of these developers are contiguous, and less affected by interruptions.

7.3.2 Time Spent Outside the IDE

Switching the context between the IDE and other applications (*i.e.,* reading e-mails) impacts the focus, flow, and productivity of a developer [LVD06, SRG15]. A developer who spends time outside the IDE, once back in the IDE, is likely to need time to “recover”: Her sessions are likely to exhibit more time spent in program understanding. Another factor that may impact the duration of understanding time is the number of such breaks: A session may end up in a “fragmented” state where the flow is frequently interrupted by context switches that lead to spending time outside the IDE, and it might have an impact on the time spent in program understanding. The number of context switches might also have an impact on the time spent in fiddling with the UI of the IDE. After a context switch, it is likely that a developer needs to re-arrange her environment to “recover” the context inside the IDE. This is what we call UI time.

To investigate these conjectures, we analyze the correlation between work fragmentation, time spent in fiddling with the UI of the IDE, and understanding time. Work fragmentation is expressed by the *time* spent outside the IDE (*i.e.,* DOI) and the *number* of times the developer goes outside the IDE (*i.e.,* OI Events). We use the Pearson Correlation Coefficient (PCC) to determine the linear correlations using the R[4] tool. Our analyses involve the 704 sessions that have time spent outside the IDE.

Time Spent Outside the IDE vs. Understanding Time

The PCC is 0.46 and is thus a weak linear correlation; using the corresponding statistical test [Tri06], we reject the hypothesis that values are not correlated at 95% confidence level with the lowest possible p-value returned by R (2.20×10^{-16}).

Number of OI Events vs. Duration of Understanding

The PCC is 0.63, and the statistical test at confidence level of 95% is in favor of rejecting the null hypothesis of non-correlation with a p-value similar to the previous test. Even if *correlation is not causation*, these findings are consistent with the hypothesis that the number of time intervals spent outside the IDE increases understanding time.

[4] See `http://www.r-project.org`

Number of OI Events vs. Duration of UI Time

The PCC is 0.65, and the statistical test at confidence level of 95% is in favor of rejecting the null hypothesis of non-correlation, with the same p-value. These results support the fact that the more context switches happen in a session, the more a developer fiddles with the UI of the IDE to recover her focus.

Table 7.5. Results – Correlation of Understanding time (UND) with the number of OI events (NOI) and the Duration of the time spent Outside the IDE (DOI)

Developer	Sess.	NOI	Avg. NOI	PCC NOI vs. UND	p-value	PCC DOI vs. UND	p-value
D1	407	2,101	5.16	0.72	2.20×10^{-16}	0.66	2.20×10^{-16}
D2	136	989	7.27	0.76	2.20×10^{-16}	0.40	1.83×10^{-6}
D3	76	154	2.03	0.47	2.27×10^{-5}	0.00	9.53×10^{-1}
D4	32	91	2.84	0.91	7.07×10^{-13}	0.80	2.89×10^{-8}
D5	19	36	1.89	0.82	2.09×10^{-5}	0.61	6.03×10^{-3}
D7	11	73	6.64	0.74	9.82×10^{-3}	0.64	3.55×10^{-2}

Table 7.5 shows the first two correlation analyses discussed above for each developer with at least 10 sessions with at least one time interval spent outside the IDE in a session (*i.e.,* D6 is not in the table because she has zero OI Events). At first sight, there is evidence of diverse developer behavior in terms of the number of time intervals spent outside of the IDE per session, which varies from a minimum of 2.37 to a maximum of 8.50 intervals per session. Results for correlation are also diverse: All p-values are very low (*i.e.,* below 4×10^{-2}) and suggest rejection of the hypothesis of non correlation. The exception is D3, for whom the duration of the time spent outside the IDE is not correlated with the duration of program understanding. However, there is a mild but significant correlation with the number of intervals spent outside the IDE. This likely means that it does not matter how much time she spent outside the IDE in total, but just the number of times her sessions are fragmented. D2 shows a similar behavior: she seems more affected by the number of times she exits the IDE rather than time spent outside. D4 is also interesting: Her sessions are not very fragmented; however, she is the developer mostly affected by the time spent outside the IDE, with strong correlation with both duration and number of intervals spent outside the IDE.

7.3.3 The Impact of the UI, Navigation, and Editing

Our data shows that on average around 14% of the time of developers is spent on rearranging the UI of the IDE, that is, resizing or dragging windows. Different experience may explain variability when aggregating data per developer. D14, for example, rearranges windows only for 4% of the time, while D2 and D12 spend around 30% of their time for this task. This might indicate that they often end up in chaotic environments [RND09] that need to be reordered or restructured.

Our data shows that pure source code navigation occupies around 3.6% of the time of developers, and that browsing occupies most of the time spent in navigation. Only 7 developers used searching. Among the people who use search, the time spent on these activities low. Presence of editing activities and editing time is also quite variable: it ranges from 1.24% for D14 to 10.67% for D7. Moreover, D7 and D16 correlate their high time spent in editing with very short time spent outside the IDE, which is probably a sign of highly focused development sessions.

7.4 Reflections

In this chapter we proposed a model that aggregates raw IDE interaction events into sprees and those into activities. From 5 million events recorded with DFLOW, the model creates 31,609 high-level development activities. We used the activities to measure the time spent by developers in 5 distinct and disjunct categories: *understanding*, *navigation*, *editing*, *UI interactions*, and *time spent outside of the IDE*.

Our results reinforce common claims about the role of program understanding by suggesting that previous research might have significantly underestimated the importance of program comprehension. In addition, developers spend 14% of their time in fiddling with the UI of the IDE, which calls for novel and more efficient user interfaces. The time spent for editing and navigating source code is respectively 5% and 4%. The large part of development is occupied by mental processes (*i.e.,* understanding) and, in the remaining time, a developer has to deal with inefficient user interfaces to read, write, and browse source code. We believe that future IDEs should tackle these problems to enable developers to focus on the tangible part of development: writing source code. We also observed that the number of context-switches (*i.e.,* times the IDE loses focus in a development session) and their duration, is linearly correlated with both the understanding time and the time spent in fiddling with the UI. This corroborates results obtained previously by researchers like LaToza *et al.* [LVD06]. Finally, the time spent outside of the IDE (ca. 8%), the frequency of such interruptions, and their subsequent negative impact on understanding, points out that developers are exposed probably too often to distractions. To draw a simplistic conclusion: Developers should not be interrupted during programming activities.

We believe that our work makes a number of contributions to the state of the art: First, with respect to the field of program comprehension, it confirms what has long been an accepted, but never validated, ground truth: program comprehension is the activity with which developers spend the vast majority of their time. The motionless staring at the screen is thus legitimized. Second, it points out that IDEs are far from perfect when it comes to the way their user interfaces are built. We believe this calls for research in novel approaches and metaphors, which so far still represent a niche research area. Third, it confirms that like many other modern workers, software developers are exposed to frequent interruptions with negative consequences.

7.4.1 Advocatus Diaboli

Dataset. We believe that our datasets are large enough to draw statistical conclusions. However, it has flaws related to the distribution of recorded sessions among the developers: More than half of the sessions come from the same developer and some developers provided us with only few minutes of interaction data. Since the last fact may influence conclusions about developer diversity, we will not consider such individuals when we reason about single developers. We included their values from completeness, but we marked them by adding a *star* ($\star$) next to their names in the Tables of this chapter. Another argument can be formulated about the missing purpose (*e.g.,* debugging, refactoring, *ex novo* implementation) of sessions. Further investigation is needed to understand how the purpose of a session and the code base and project size influence studies like the one we propose.

Inference Models. We inferred activities starting from low-level events like keyboard and mouse sprees, and meta-events from the IDE like saving a method or inspecting a field of an object. Recording low-level events minimizes the possibility that we discard relevant events and do not capture exact duration of activities. However, since we may not monitor every possible meta event

of the IDE (*e.g.,* special ad-hoc plugins and widgets) we may potentially interpret some activity in the wrong way. To cope with this threat, we made sure that all developers used the standard widgets of *Pharo* for which our model correctly classifies events and sprees in the correct class of activities. As future work, we plan to cross-validate our automated activity extraction with concrete observations (*e.g.,* think-aloud) to understand to what extent the extracted activities match the actual activities.

The same applies with basic understanding. In principle, the fact that small periods of idleness (inter- and intra-activities) are mapped to program understanding is an explicit assumption that we made, but indeed they could be mini interruptions unrelated to development, like the programmer checking his phone. However, the reverse critic could be done to some of the moments spent outside the IDE. They could be timespans spent in checking documentation or other development artifacts supporting program understanding, that are completely absent from our model. We still need cross-validation to ensure that our interpretation is correct, but we believe that the issues above compensate themselves and do not invalidate our measurements involving program understanding.

Results. The results of the correlation analyses summarized in Table 7.5 are consistent with the fact that the time outside the IDE influences the total program understanding time. However, the dynamics of development are complex, and other factors influence the duration of program understanding. For example, developer experience on the task at hand may strongly decrease the impact of session fragmentation on program understanding. Moreover, the extent and impact of fragmentation depends on the specific activity performed outside the IDE – it is likely that a chat on an unrelated matter or browsing a social network's feed may impact more than reading a related Stack Overflow discussion.

7.4.2 Wrapping Up

In the last two Chapters we discussed how to use IDE interactions to estimate how developers spend their time. The next Chapter, instead, focuses a single development activity: source code navigation. We present an approach to model and measure how efficiently developers navigate source code in the *Pharo* IDE using interaction data.

8

Measuring Navigation Efficiency in the IDE

Developers spend more than 40% of their time navigating and reading source code [SLVA97]. In the previous chapter we have seen that we can leverage interaction data to provide an estimate to the time spent in program comprehension activities. According to our results, these activities potentially absorb up to 70% of the time of developers [MML15b]. During software maintenance another key activity is source code navigation [RCM04, PFS+11]. Researcher observed that navigating code is essential to support the construction of *mental models* of the system [SFM99, LVD06], *i.e.,* mappings between relevant parts of source code and the purpose of the program [SFM99], which in turn is a prerequisite for software comprehension.

To navigate and read source code, developers leverage the user interfaces offered by an IDE. Examples include code browsers, package explorers, senders (*i.e.,* callers) browsers, or implementors (*i.e.,* definitions) browsers. By construction, these UIs force developers to perform a certain number of interactions with the IDE to navigate source code: The *Eclipse*'s *Package Explorer*, for example, forces developers to follow structural relationships between code entities, introducing potentially unnecessary navigations. Moreover, a given programming task may require the programmer to navigate more entities than needed.

While there is limited evidence that this navigation process is inefficient (*e.g.,* [RCM04, MKF06, KMCA06]), it is largely unclear how to actually define, and also measure, the navigation efficiency. In particular, it is non-trivial to define an optimal navigation scenario for a given task which consists of a set of navigated entities.

In this chapter we present a series of models that use IDE interactions to estimate the navigation efficiency of developers inside the *Pharo* IDE. We call *navigation efficiency* the ratio between an ideal scenario and the actual behavior of the developer. Having both a precise definition of navigation efficiency as well as knowing the optimal navigation scenario are prerequisites to obtain an in-depth understanding of how IDEs can be made more efficient from a user interface perspective, when it comes to supporting software development activities.

Structure of the Chapter

Section 8.1 explains the mechanics of navigation in the *Pharo* IDE and summarizes the dataset of this study. Section 8.2 introduces the concepts we used to model navigation efficiency with IDE interactions. Section 8.3 illustrates our preliminary model for navigation efficiency and our preliminary results. Section 8.4 explains the refinements to the model and details the updated results. Finally, in Section 8.5 we discuss the results and Section 8.6 concludes the chapter.

8.1 Source Code Navigation in the Pharo IDE

Navigating source code inside an IDE is supported and influenced by the components and by the UIs that it offers. This section describes the classical way developers navigate source code in the *Pharo* IDE and summarizes the dataset of this study.

8.1.1 Structural Source Code Navigation in *Pharo*

Pharo is a window-based IDE (as opposed to tab-based IDEs, like *Eclipse*). As described in Section 3.2, its object model is composed of methods, protocols, classes (instances of metaclasses), and packages. Methods, classes, and packages are well known concepts shared with many other object-oriented languages. Protocols are essentially labels to group methods according to their intent as defined by the programmer or by conventions (*e.g.,* the *"accessing"* protocol groups all the accessors of a class).

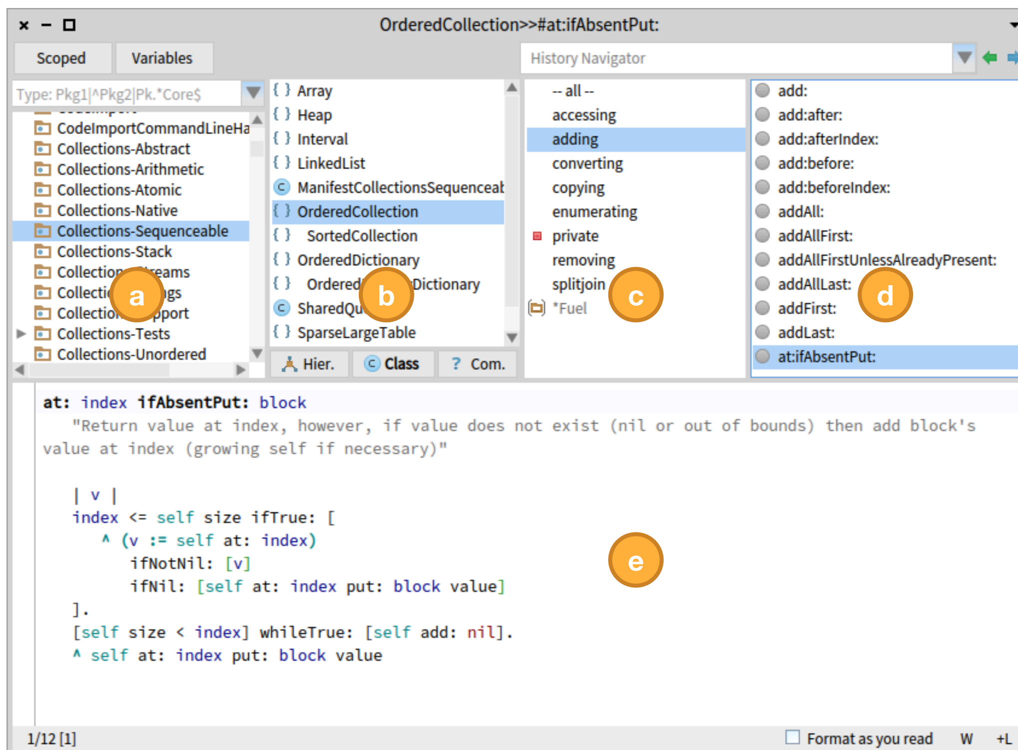


Figure 8.1. The Code Browser: The main UI to Navigate and modify code in the *Pharo* IDE

To navigate and/or modify source code, the main UI offered by the *Pharo* IDE is the *Code Browser*, depicted in Figure 8.1. The upper part of the browser depicts a tree view for the structure of the code featuring 4 columns, one for each element of the object model, *i.e.,* (a) packages, (b) classes, (c) protocols, and (d) methods. To select an entity of interest, developers follow the structural organization of the source code. In the browser of Figure 8.1, for example, the developer selected the method `at:ifAbsentPut:` in the protocol *"adding"* of class `OrderedCollection` which is contained in the package `Collections-Sequenceable`.

Example Navigation Scenario

To navigate to a specific method, the developer has to perform the following steps:

1. **Select a Package.** The leftmost column of the browser (Fig. 8.1.a) lists all the packages. The developer browses them and clicks on the one she wants to explore.

2. **Select a Class.** After selecting a package, *Pharo* populates the second column of the browser with the list of all the classes contained in that package (Fig. 8.1.b). The developer can now choose the one she is interested in.

3. **Select a Protocol (optional).** After selecting a class, *Pharo* populates the third column (Fig. 8.1.c) with all the available protocols and the fourth column (Fig. 8.1.d) with all the methods of the class (regardless of the protocol). The selection of the protocol is optional: A developer can also directly browse the complete list of methods.

4. **Select a Method.** If a developer selected a protocol, the last column of the browser will only list the methods belonging to that protocol (Fig. 8.1.d).

Finally, once the developer selects a method, the bottom half of the browser (Fig. 8.1.e) displays the source code of the selected method, and lets the developer read it and modify it.

In a nutshell

The code browser forces developers to follow structural source code relationships. While the developer often knows which place to reach, she is forced to perform a series of UI interactions to do so. In tab-based IDEs like *Eclipse*, even if the mechanics are slightly different, often developers are in the same situation, *e.g.,* when using the *Package Explorer*.

8.1.2 Dataset

Table 8.1 summarizes the dataset for the current study. It counts 711 sessions coming from 6 developers. Development sessions do not include interruptions lasting more than 5 minutes (*i.e.,* considered inactivity) [MML15b].

We consider a total of 20 days (*i.e.,* 474h 6m 20s) of development time. On average, a session lasts for about 40 minutes. However, if we observe the quartiles[1], there is some variability: Half of the sessions (Q_2) are shorter than 30 minutes and one fourth of the sessions (Q_3) are actually longer than 53 minutes. The first quartile, instead, is around 19 minutes.

Table 8.1 also reports details on the number of meta events and program entities involved in the sessions. On average, each session features about 280 meta events, of which around 217 are navigations, 18 are edits, and 52 are inspects. If we look at the program entities involved, on average developers deal with 35 entities per session. They only edit around 6 entities per session. The remaining entities are only navigated (ca. 29) or subject to inspection (ca. 7).

8.2 Modeling Navigation Efficiency with Interaction Data

We call *"navigation efficiency"* the ratio between an *ideal navigation scenario* and the *real navigation effort* spent by developers [MML16a]. The *real effort* corresponds with the number of

[1] In all the Tables, Q_n indicates the nth quartile.

Table 8.1. Dataset – Study on the Navigation Efficiency

General				
Number of Sessions				711
Number of Developers				6
Total Duration				474h 06m 20s

Average Duration	Q_1	Q_2	Q_3	Avg.
Session Duration	18m 40s	30m 30s	53m 24s	40m 01s

Meta Events (Session)	Q_1	Q_2	Q_3	Avg.
Navigation	57	124	261	216.87
Edit	4	7	13	18.17
Inspect	2	14	45	51.78
All	**88**	**179**	**362**	**279.41**

Entities (Session)	Q_1	Q_2	Q_3	Avg.
Navigated	10	20	40	28.85
Edited	2	4	6	5.47
Inspected	1	5	11	7.29
Total	**15**	**26**	**44**	**34.85**

navigation events performed by a developer in a development session. This data is contained in the IDE interaction histories recorded with DFlow.

The *ideal effort*, instead, expresses the *theoretical minimum* number of navigations that a developer has to perform in a session to accomplish her tasks. It is an estimate that depends on two main variables: the *program entities involved* and the *cost* of navigating such entities.

Involved Program Entities

During a development session, developers interact with different program entities. Some of them are modified, others are only navigated, for example to build a mental model of the system, *e.g.,* [SFM99]. In a completely, yet unrealistic, ideal case a developer visits *once* each of the entities that she needs to change as part of her task. We call those entities *edited entities*. In the *real* scenario, *i.e.,* the recorded interaction history, a developer navigates more entities than the ones just edited (*e.g.,* [DCR05, YR11]): We call those entities *touched entities*. The touched entities contain, in addition to the edited entities, all the entities that are essential for the construction of the mental model, and also spurious entities, *e.g.,* entities that were navigated by mistake by the developer and do not contribute to program understanding.

Since it is almost impossible to identify spurious entities, our model for the ideal navigation effort only considers the edited entities. However, even if we only consider the edited entities to represent a programming task, one may take into account the fact that developers may visit entities more than once and/or in a specific order.

Navigation Cost

The navigation cost is expressed in terms of the number of UI interactions (*i.e.,* clicks) needed to reach an entity of interest. Hence, the cost is strictly related to how the UI that the developer uses to navigate, and modify, source code supports the navigation.

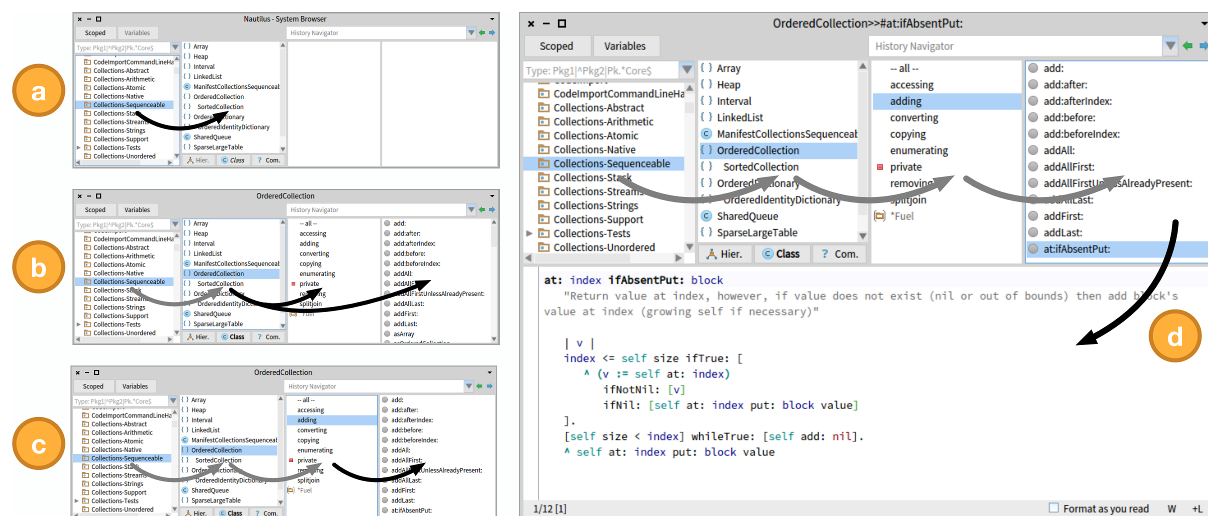


Figure 8.2. Navigating source code in the *Pharo* Code Browser: (a) Selecting a Package, (b) a Class, (c) a Protocol, and (d) a Method

For example, we can compute the cost of navigating to an entity with the *Pharo* Code Browser, a tool similar to the *Package Explorer* in *Eclipse*. Figure 8.2 shows all the steps needed to navigate to a method. With one IDE interaction the developer selects a package and obtains the list of its classes (Figure 8.2.a). With another click she can select a class and obtain the list of protocols contained in it (Figure 8.2.b). At the same time the browser populates the list of all methods. With the next click she can select a protocol and obtain the list of methods, or directly selecting one of the methods from the rightmost column (Figure 8.2.c). Finally, she selects a method and obtains its source code in the browser (Figure 8.2.d).

We define the navigation cost (NC) for an entity x as:

$$NC(x) = \begin{cases} 1 \textit{ if } x \textit{ is a package,} \\ 2 \textit{ if } x \textit{ is a class,} \\ 3 \textit{ if } x \textit{ is a protocol,} \\ 4 \textit{ if } x \textit{ is a method.} \end{cases}$$

However, the code browser is not the only way to navigate code. As all modern IDEs, the *Pharo* IDE also features different UIs that support navigation (*e.g.,* search facilities). This is one of the core aspects described in the following section when we describe how we incrementally build our models for navigation efficiency.

8.3 A Naïve Model for Navigation Efficiency

We developed a series of models for navigation efficiency, starting from a naïve model and refining it to mimic more realistic scenarios. Our initial model considers two variants for the involved entities and two navigation cost models [MML16a].

Involved Program Entities

In an ideal, yet unrealistic, scenario, a developer knows exactly what she needs to modify and does not need to construct a mental model. Thus she does not visit entities that she do not need

to modify. The navigation efficiency is thus computed only considering the set of entities that have been *edited* during the development session. The model considers two possible alternatives:

WORKING SET. It represents the more ideal case: The developer knows exactly what she must edit, thus she only visits each entity exactly once to perform the needed modifications, never touching the same entity again.

WORKING SEQUENCE. In a more realistic case, the model considers the sequence of edited entities, as they appear in the recorded interaction history. This estimate gives importance to the so called *revisits*, *i.e.,* when a developer navigates, and also edits, more than one time the same entity during a development session [SKG+13]. During re-factoring, for example, a developer re-edits previously modified entities. In *Test Driven Development (TDD)*, multiple edits are interleaved with the writing of test cases to support the implementation of a feature.

Navigation Cost

The navigation cost expresses the price of navigating to an entity. We define the cost in terms of the number of navigation events (*i.e.,* clicks) that the developer has to perform to reach an entity [MML16a].

We identify two cost models:

UNITARY. In the most ideal case the developer can directly jump to an entity, *e.g.,* by using a search interface.

MAX. In the worst case, instead, the developer uses a new code browser each time she has to perform a new navigation. In this case, the navigation cost is exactly the one imposed by the code browser.

Results

Table 8.2 summarizes our results by applying this naiv̈e model. It reports four alternatives of navigation efficiency by varying the two parameters of the model.

Table 8.2. Results – Preliminary estimates for Navigation Efficiency

	Working Set				Working Sequence			
Cost	Q_1	Q_2	Q_3	Avg.	Q_1	Q_2	Q_3	Avg.
Unitary	0.018	0.032	0.060	0.051	0.036	0.060	0.112	0.096
Max	0.073	0.129	0.238	0.206	0.137	0.242	0.450	0.387

In the more realistic case (*i.e.,* working sequence), developers perform on average 1.5x to 9x more navigations than in the ideal settings[2]. Observing medians, unneeded navigations range between 3x to 16x more than the ones ideally required. If we compare the real effort with the most ideal scenario, the working set, the results are even worse, with 4x to 19x more navigations. Medians show that in 50% of sessions developers perform more than 30x the needed navigations.

The next section details our more precise model called *"Delta-cost"* (or Δ-cost).

[2]Redundant navigations are computed as follows: $\frac{1}{Efficiency} - 1$.

8.3.1 A More Realistic Cost Model: The Δ-cost

The naiv̈e model considers the two extreme navigation costs: UNITARY and MAX. The former assumes that developers jump to the entity of interest with one single action, while the latter assumes that they always use a new code browser and perform the entire navigation chain.

Since we believe that the truth lies somewhere in between, we introduce the *delta-cost* (or Δ-cost) model that better matches the reality of navigation in the code browser. In more realistic settings, a developer neither always directly jumps to the entities nor opens a new code browser every time. What she does is "reuse" part of the code browser's UI.

Navigating to a new entity thus depends on the currently selected entity. For example, suppose we have a code browser with the method `bar` of class `Foo` selected. If the developer wants to visit a new method of the same class. The theoretical cost (*i.e.,* or MAX cost) would be 4, but with the Δ-cost model, we can subtract the cost the developer has already payed (*i.e., "navigation prefix"*, the cost of navigating to the common part of the navigation chain, in this case the cost of visiting the class `Foo`, that is 3). The resulting navigation cost is $4-3=1$.

Clearly, with the Δ-cost the navigation order is essential. Also with this model, when compute the navigation effort we have two scenarios for the involved entities:

WORKING SEQUENCE. The working sequence considers the entities as they appear in the developer's interaction history. Thus, it is enough to apply this new cost model without any modification.

SORTED WORKING SET. Since the cost of navigating an entity depends on the currently selected entity, and we aim at an ideal scenario, simply removing duplicates from the sequence is suboptimal. Thus, we consider the set of edited entities in the order that minimizes the total navigation cost. This model assumes a more ideal case than the working sequence: We group navigations from methods belonging to the same class, and from classes belonging to the same package. In practice, we take the sequence of all the edited entities, we remove the duplicates, and we sort them as per Figure 8.3.

Original Sequence	Set	Sorted Set
P1.Foo	P1.Foo	P1.Foo
P1.Foo>>#m1	P1.Foo>>#m1	P1.Foo>>#m1
P1.Baz	P1.Baz	P1.Foo>>#m2
P1.Foo>>#m1	P1.Foo>>#m2	P1.Baz
P1.Foo>>#m2	P1.Baz>>#m3	P1.Baz>>#m3
P1.Baz	P1.Baz>>#m4	P1.Baz>>#m4
P1.Baz>>#m3		
P1.Baz>>#m4		

Figure 8.3. Sorting entities to minimize the Δ-cost

In Figure 8.3 the notation `P1.Foo»#m2` represents the method `m2` of class `Foo`, contained in package `P1`. The first column of the figure shows the original sequence of edited entities, while the central column shows the corresponding set, with duplicate entities removed: In the example of Figure 8.3, we remove `P1.Foo»#m1` and `P1.Baz`.

Thus, in the second step we sort the entities according to their common *"navigation prefix"*. In the original interaction history, the user initially edits the method `P1.Foo»#m1`, then the class `P1.Baz`. Afterward she returns to a method of class `P1.Foo` and finally edits two methods of class `P1.Baz`.

This sequence is not optimal in terms of Δ-cost because navigating multiple times from `P1.Foo` to `P1.Baz` and viceversa leads to a higher navigation cost as opposed to perform this navigation once. To minimize the navigation cost, we put as close as possible entities with the longest common *"navigation prefix"*, *i.e.,* in this case we move the navigation to `P1.Foo»#m2` close to the navigation to `m1` of the same class. The result is a sequence of entities with a minimal navigation cost (*i.e.,* Δ-cost) across adjacent pairs of entities.

This choice of ordering the edited events and the use of the Δ-cost model, idealizes a process in which the developer has a solid mental model for the current editing task, and she is able to identify each method to be edited in each class (and each class to be modified in a specific package), without visiting an entity more than once in a session.

Table 8.3 summarizes the navigation efficiency measured with the Δ-cost model.

Table 8.3. Results – Navigation Efficiency with Δ-cost

	Efficiency (Δ-cost)			
	Q_1	Q_2	Q_3	Avg.
Sorted Working Set	0.028	0.062	0.116	0.094
Working Sequence	0.039	0.086	0.154	0.126

In the most ideal case (*i.e.,* sorted working set) the efficiency of developers is on average 0.094, *i.e.,* they perform 10 times more navigations than needed. The situation is better if we consider the actual sequence of edited entities, *i.e.,* developers perform 7 times more navigations than needed.

8.3.2 Limitations

Developers' Diversity and Demographics

Our dataset includes sessions coming from 6 developers. To present the data from a different perspective, we observed how the navigation efficiency varies across different developers. One hypothesis is that navigation efficiency might be connected to a developer's expertise, or with the developer's knowledge of the system.

Figure 8.4 shows a box-plot of the navigation efficiency per developer using the Δ-cost and the sequence of edited entities.

D2 and D3 are inexperienced with the *Pharo* IDE. In addition, D2 is also new to programming, while D3 is a graduate student with 4 years experience with other IDEs. The remaining developers, instead, are professional developers familiar with the *Pharo* IDE: They have been developing in *Pharo* for more than 5 years. Inexperienced developers are not worse source code navigators. As a matter of fact, D2 is outperforming the more experienced developers.

How can this be? Moreover, in some sessions the efficiency of D2 is greater than 1, meaning that she is outperforming even the ideal model. *How is this possible?* The answer lies in a key limitation of our model: Neglecting the fact that developers often do not edit the source code with the code browser, but by using other UIs of the *Pharo* IDE such as its *live debugger*.

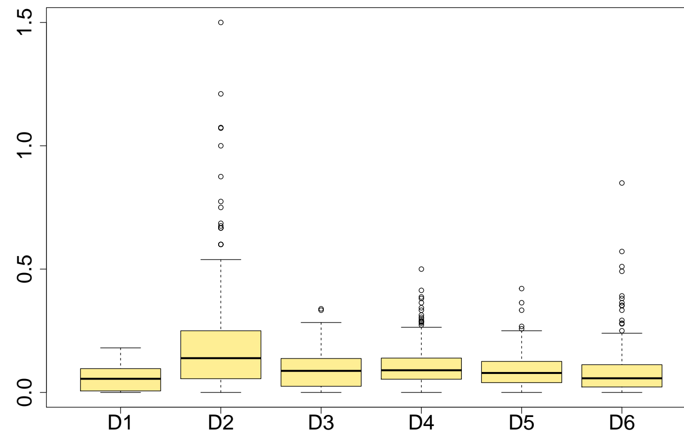


Figure 8.4. Navigation Efficiency per developer [Δ-cost + sequence]

More than a Debugger

Figure 8.5 shows the UI of the *Pharo* Debugger which is composed of: a stack trace (a), a source code editor (b), and a variable inspector pane (c).

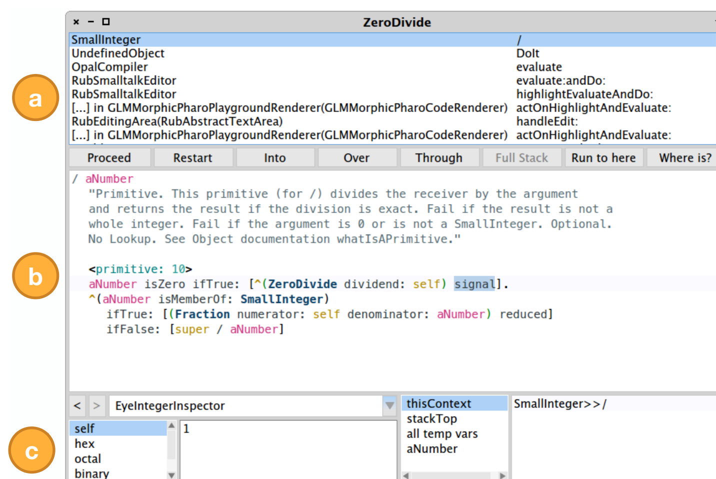


Figure 8.5. The *Pharo* Debugger UI

In *Pharo* developers can use the editor pane of the debugger (Fig. 8.5.b) to modify the code of the selected method on the execution stack (Fig. 8.5.a). After saving modifications, the IDE will re-compile the new method, and resume execution.

Moreover, the stack trace (Fig. 8.5.a) is *navigable*. Thus, developers can select an entry from the stack trace to obtain its source code in the editor pane (Fig. 8.5.b). Afterwards, the developer can modify it and save the changes, with no difference with respect to the code browser. Thus, limiting the scope of navigations to the code browser is too simplistic, which calls for an additional refinement of our model.

8.4 A Refined Model for Navigation Efficiency

Building on top of the limitations of the naïve model described in the previous section, we refine our model by including navigations that happen in UIs different than the code browser. In turn, we refine both the scenarios for real and ideal navigation measures.

8.4.1 Navigation Beyond the Code Browser

Source code navigation happens in at least three other UIs, namely the Debugger, the *Senders UI* and the *Implementors UI* (*i.e.*, similar to the *"Jump to declaration/definition"* of *Eclipse*).

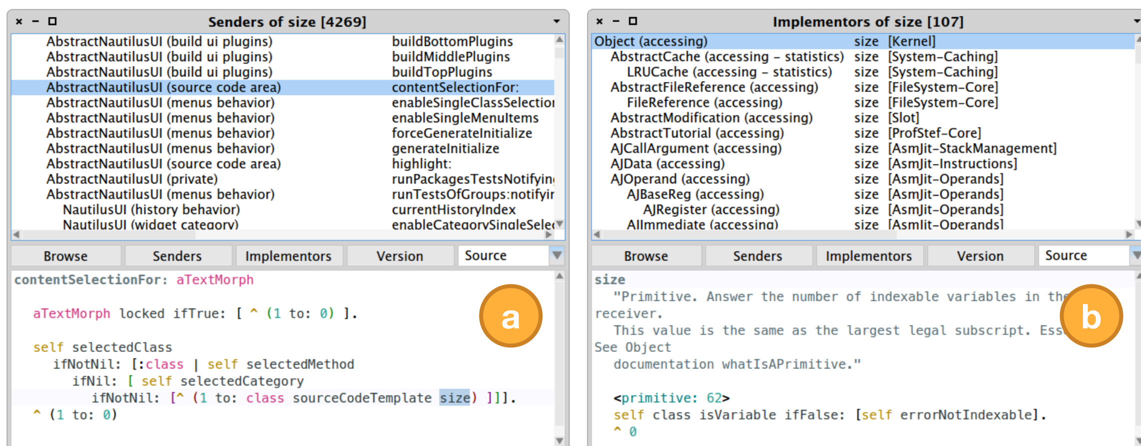


Figure 8.6. The Senders UI (a) and the Implementors UI (b) for the method `size`

Senders UI. In *Pharo* developers can select a method and ask for all the methods in the system that invoke that method. The IDE will open a *senders UI*, as the one depicted in Figure 8.6.a. The name of this UI, "senders", comes from the fact that *Pharo* follows a *messaging* paradigm, thus invoking a method means sending a message to a given object.

Implementors UI. A complementary UI is the *Implementors UI* that, given a method name shows all methods in the system with this name (belonging to different classes). Figure 8.6.b shows an example of this UI.

Both UIs offer a list of methods on the topmost half and a code editor pane on the bottom. The top part is *navigable*, thus lets developers directly navigate entities, *i.e.*, with unitary cost.

8.4.2 Refining Real Navigation

Figure 8.7 shows the updated box-plot for navigation efficiency when considering the navigations happening in other UIs, *i.e.*, debugger and senders/implementors UIs.

Compared with the previous results depicted in Figure 8.4, now the efficiency of D2 is similar to the one of the others developers. However, now the *ideal navigation cost* is wrong because it considers all the navigations happening in the code browser.

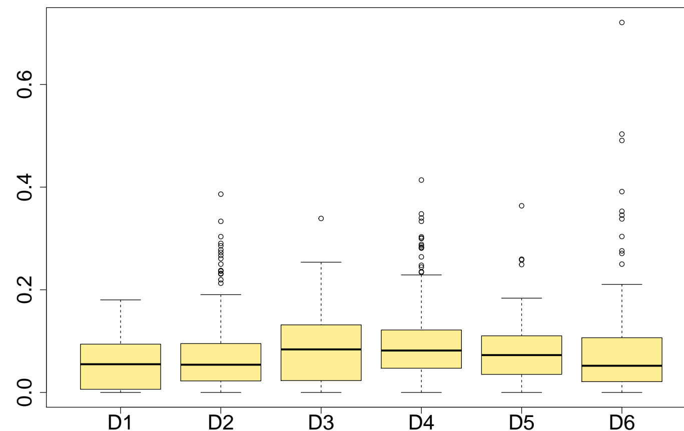


Figure 8.7. Navigation Efficiency considering Navigations outside the Code Browser

8.4.3 Refining Ideal Navigation: The UI-Aware Model

Navigations happening outside the code browser are more efficient than navigating with the code browser, *i.e.,* list vs. tree navigation. Thus, it is reasonable to assume a unitary cost, as in each of the four selections needed in the code browser. This models the situation when the developer looks at the list of methods, for example in a debugger, and clicks on a method.

This assumption can be applied to refine the model that considers the sequence of edited methods. In the other case (*i.e.,* the *set* of edited methods) if a method is edited in both a code browser and a debugger, it would be rather arbitrary to choose one cost model with respect to the the other to compute its ideal navigation cost.

The refined model computes what we called *"UI-Aware Navigation Cost"* as follows:

- It considers the sequence of edited entities as they appear in the recorded session;
- For each edit event, it checks in which UI component it has been performed:
  - For events in the code browser, it uses the Δ-cost model;
  - For events in other UIs,[3] it considers a unitary cost.

Rationale

The *UI-Aware* model takes into account the fact that some tasks, like debugging, are performed in different UIs that support different relations between code entities, *e.g.,* runtime sequence of method calls. It would be imprecise to assume that in these cases the developer should ideally reconstruct the same context, *i.e.,* the same relationship between code entities, by using the Δ-cost model, as she should be forced to use a code browser.

The UI-Aware cost model is *always* less or equal than the one computed using the Δ-cost on the sequence of edited entities. In fact, being the sequence identical, it may only reduce the total number of ideal navigations by assuming a unitary cost for a subset of the edited entities.

For instance, consider the developer inspecting the stack in the debugger. The developer's inefficiency, in this case, effectively models the spurious navigations that are needed to understand

[3] Such as the Debugger, Senders/Implementors UI, and Search UI.

where the given edit must be performed with respect to the actual location where the execution has been stopped. Another example is the user trying to search for a specific implementation of a given method. The wasted navigations performed in a implementors browser represent the ones required by the specific search of the desired implementation.

8.4.4 Results

Table 8.4 shows the navigation efficiency computed by applying the UI-Aware cost.

Table 8.4. Results – Navigation Efficiency with UI-Aware navigation cost

	Efficiency (UI-Aware)			
	Q_1	Q_2	Q_3	Avg.
Working Sequence	0.025	0.056	0.098	0.073

With this model, the navigation efficiency is, on average, 7.3%, with a median of 5.6%. At first sight, it might look the refinements did not change much from our very first observation, given that the results are in the same order of magnitude. However, the new model is now robust with respect to the contradictions that were raised before.

8.5 Reflections

8.5.1 Developer diversity

Figure 8.8 shows the values of navigation efficiency using the UI-Aware cost model. All developers seems to be equally efficient. We applied the *Mann-Whitney-Wilcoxon Test* for each pair of developers to find any possible difference between developers. Since for each pair we obtain p-values greater than 0.05, there is no evidence that the efficiency follows a different distribution between developers.

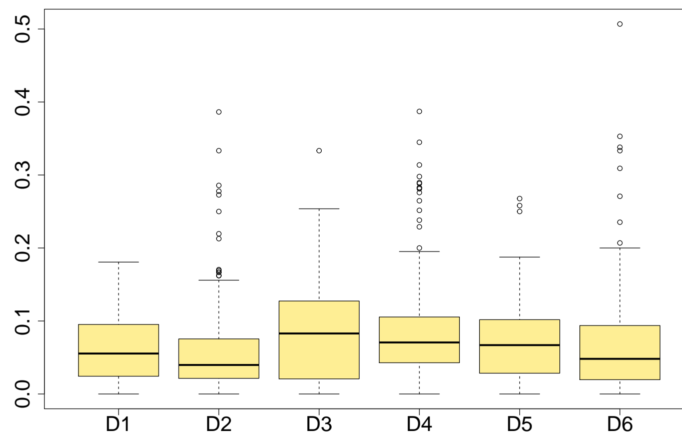


Figure 8.8. Results – UI-Aware Navigation Efficiency per developer

8.5.2 Outliers

While on average efficiencies are very low, there are many outlier sessions with higher efficiency (*i.e.,* greater than 0.2). A conjecture is that, for tasks where developers have a consolidated mental model, navigations are more focused. We leave the investigation of such sessions as a future work.

8.5.3 Significance of Edited Entities

Our models of navigation effort (thus, efficiency), rely on the fact that a task can be characterized by only the edited entities. However, programming requires the construction of mental models that inevitably mandate visiting some entities that do not need changes.

We justified the adoption of edited entities as a reference because it is very hard to distinguish the navigated entities that are required for the mental model construction from the ones that are not. To depict the opposite upper bound, we recompute the results by considering all the entities involved a session (*i.e.,* regardless of the fact that they were edited or not). Such an upper-bound considers every navigated entity as essential for the construction of the mental model.

The real navigation efficiency lies between the ideal model (see Table 8.4) and this upper-bound (see Table 8.5).

Table 8.5. Results – An upper bound for Navigation Efficiency: UI-Aware Navigation Efficiency considering all the entities involved in a development session

	Efficiency (UI-Aware)			
	Q_1	Q_2	Q_3	Avg.
All Touched Entities Sequence	1.073	1.22	1.43	1.26

8.5.4 Contradicting Findings?

Better developers are *not* the better navigators. Intuitively experienced developers should be better navigators. However, in our study novice developers (*i.e.,* D2 and D3) perform similar to experienced developers. Our conjecture is that the navigation cost imposed by the UI of the IDE cannot be avoided, and all developers have to use it in a similar way. The IDE might sustain less experienced developers, but at the same time it does not fully support more experienced ones.

How can efficiency be higher than 1? Modern IDEs include facilities that allow for "super-human efficiency". A simple "rename refactoring", for example, can potentially change hundreds of entities with one simple click. Currently, our model does not take that into consideration, and would model the ideal navigation costs as hundreds of (un)necessary navigations.

8.5.5 Threats to Validity

Construct Validity. Our definition of navigation efficiency assumes that the edited entities are enough to characterize a development task. This assumes an *ideal* case where developers have a perfect mental model of the software system and a clear task. This scenario is likely unattainable in practice, due to factors such as the high complexity of source code and its comprehension. However, this measure serves as a baseline to understand how the reality differs from this ideal case. To mitigate this threat and understand how the efficiency changes when considering all the entities, regardless of the fact they were edited or not, we computed the navigation efficiency

using all the entities appearing in a development session (see Table 8.5). This represents the opposite, equally unrealistic scenario, where the developer navigates a set of entities which is significantly bigger than the ones ideally required to implement the task, including the entities needed to construct the mental model. Thus, we believe that Tables 8.4 and 8.5 provide lower and upper bounds for the navigation efficiency. A realistic measure of efficiency lies in this range and depends on a number of factors such as which entities are necessary to perform a given programming task, including the difficulty and the nature of the task itself.

Another threat consists in the fact that our navigation model only accounts for list selections, while a developer might also have to scroll through a list prior to performing a selection. Even though this is essentially true, when we compute the efficiency we compare the ideal cost of selecting the entities with what we call "navigation events". Indeed, in our interaction histories these events only represent list selections, thus the comparison is adequate, and consistent with our final goal: Understanding to what extent an ideal scenario differs from reality.

Internal Validity. Our results suggest that developers are inefficient at navigating source code. In addition to the navigation cost imposed by the UI of the IDE, other factors may hinder the navigation efficiency of developers. For example, the expertise of the developer and the difficulty and the nature of the task at hand might have an impact. Even thought we did not focus on the causes for such inefficiency, Figure 8.8 provides preliminary evidence for which the level of expertise seems not to have an impact on the navigation efficiency. However, as part as our future work, we will study, and measure, the impact of other factors on the navigation efficiency.

In our work we considered five UIs: Code Browser, Debugger, Search, Senders, and Implementors UIs. To the best of our knowledge, these are the main UIs developers use to navigate source code in the *Pharo* IDE. However, a small part of the navigation might involve interactions with other UIs.

External Validity. Our results refer to the *Pharo* IDE. In spite of the fact that our model can be easily adapted to other IDEs and other UI metaphors (*e.g.,* tab-based environments), specific data modeling and evidence is needed to generalize our results to other programming environments. However, despite of the fundamental differences between various IDEs, we believe that also tab-based IDEs are poor vehicles to support the navigation of source code.

8.6 Summing Up

Leveraging IDE interaction enabled us to better understand the mechanics of the exploration process in the *Pharo* IDE. The raw data, however, needs to be modeled and interpreted to provide meaningful insights. This chapter presented a series of models to estimate the navigation efficiency of *Pharo* developers from raw IDE interaction histories. Our contribution includes a definition of navigation efficiency and an incremental construction of optimal navigation scenarios. Our results provide preliminary evidence that *Pharo* developers are inefficient when navigating source code. We believe that *Pharo*, together with all mainstream IDEs, are poor vehicles to support modern software development being nothing more than glorified text editors [Nie16]. In this regard, there is a great potential in the investigation of new, different UI paradigms for software development, some of which have already achieved noteworthy results, *e.g.,* CODE BUBBLES [BZR+10], DEBUGGER CANVAS [DBR+12], and web IDEs, such as CLOUD9 and CODIO.

Part III

Visual Analytics of Development Sessions

9

Understanding How Developers Use the User Interface of the IDE

The user interface of an IDE offers a large number of facilities to manipulate source code, such as inspectors, debuggers, recommenders, alternative source code viewers, *etc.* The *Eclipse* IDE, for example, offers *perspectives* (*i.e.,* visual containers for a set of views and editor) customized for different tasks such as *developing*, *debugging*, or *running test suites*. Murphy *et al.* studied how developers use these perspectives: They found that programmers use most perspectives offered by the IDE to varying degrees and that they often use keyboard shortcuts to perform activities [MKF06]. Besides this study, it is largely unclear how developers use the UI of IDEs and whether they give appropriate support to developers.

In this chapter we present a visual approach to answer the question: "*How do developers use an IDE with respect to the user interface it offers?*." Our approach leverages IDE interaction data recorded by DFlow to produce an interactive visualization with two different purposes. On the one side it shows how developers interact with UIs, and how the work is essentially organized in development tracks, with each track led by a main window. The view depicts how developers alternate between such different tracks during activities, and how the environment grows and shrinks from a UI point of view, giving a visual representation of the environment's "entropy". On the other side, the view synthesizes how development activities relate to UI usage, enabling the understanding of the UI structure at important development events like source code edits and commits. This combined view helps to gather a better understanding of the data being visualized. For example, it is interesting to see what happens to the source code when the developer spawns multiple windows and how the number of active windows influences the navigation between source code elements.

The visual analysis led to the development of a pattern language to characterize both developers and session types. For example, we identified "*conservatives*" developers that use to use a limited number of windows and, on the other side, "*frenetic*" developers that continuously spawn windows, most likely due to a complete immersion in the development, *i.e.,* a state of mental focus so intense that awareness of the real world is lost [LHB03], also known as "flow" [Csi90].

Structure of the Chapter

Section 9.1 illustrates our approach and presents the principles of our visualizations. Section 9.2 uses the visualization to tell development stories on our dataset. Section 9.3 presents a categorization of developers and sessions using the findings from our visual analysis. Finally, Section 9.4 sums up and concludes the chapter.

9.1 Visualizing UI Usage: Principles and Proportions

Figure 9.1 details our visualization to depict how developers use the UI of the *Pharo* IDE. The view is composed of two parts: an *UI View* and an *Activity Timeline*.

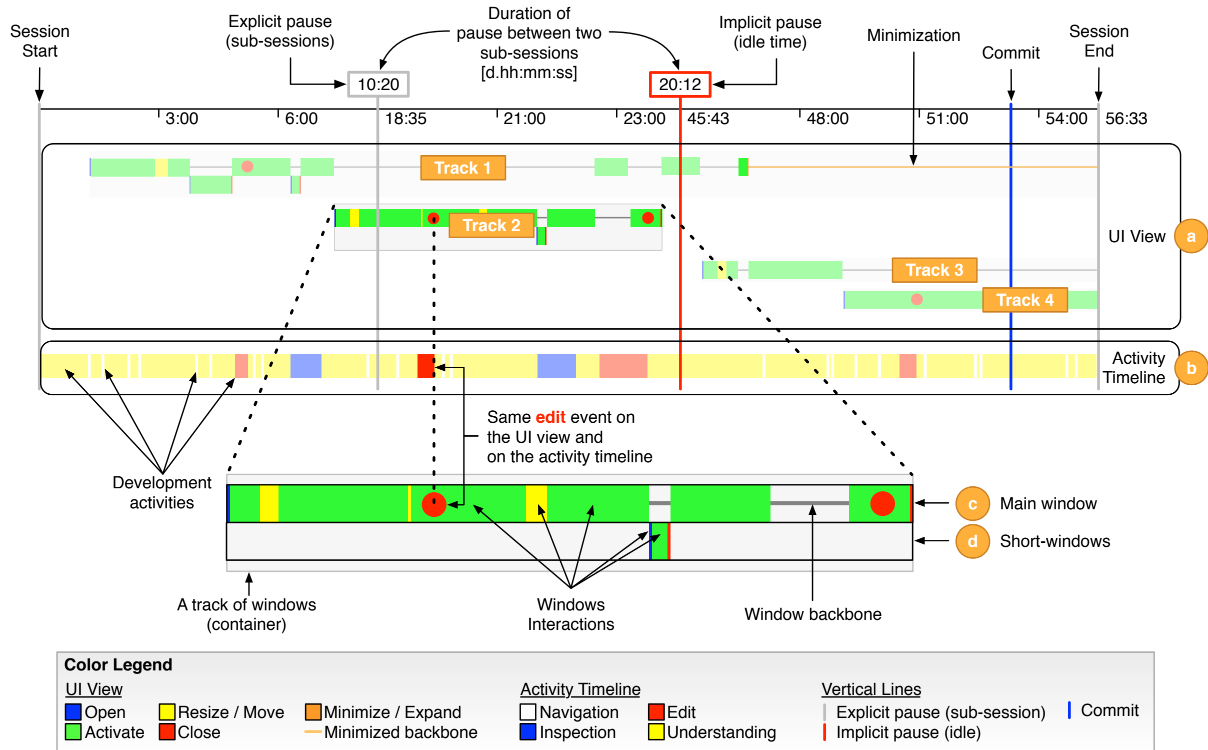


Figure 9.1. Principles and proportions of the visualization

UI View

The *UI View* (Figure 9.1.a) depicts the interactions of the developer with the UI of the IDE, that is composed of multiple windows. This part of the visualization identifies and summarizes the different *tracks of windows* that the developer follows while working. A track is a composition of a main window with a set of associated short-lived windows.

Tracks of windows, essentially, represent where and how the developer used the UI elements of the IDE. There are developers that concentrate their work on a single track and developers that are more proficient when spreading their work on multiple parallel tracks. Each track of windows is "dominated" by one window, i.e., the main window of the track (*e.g.,* Figure 9.1.c for track 2). In turn the main window might have a number of short-lived windows associated to it, the *short-windows* (*e.g.,* Figure 9.1.d for track 2). These short-windows are windows with a lifespan (*i.e.,* the time that spans from their open to their close time) shorter than a given threshold (default: 1 minute). To determine which is the main window originating a small-window, say W_S, we search, among main windows, which one was last active before the birth of W_S and is still open during its lifetime. Once we created all the associations between main and small-windows, we apply the time-based horizontal layout: The horizontal coordinate of each container, or track, represent the open time of the main window that dominates it. The width of the container is proportional to the lifespan of the main window. The height of the container varies according to

the number, and position, of short-windows. In the general case all short-windows of a track are positioned at the same y-coordinate. However, in case of two overlapping small-windows (*i.e.,* their lifespans overlaps), the layout pushes the second small-window down. The layout considers the windows in order of appearance, *i.e.,* sorted by open time. The y-coordinate of containers is used only to avoid overlapping between them.

A window is represented by a line, the *window backbone*. The length of this line is proportional to the lifespan of the window that represents. Window interactions, *i.e.,* events, are positioned on this line. Each event is a box with fixed height. Its length is proportional to the duration of the event. The x-position of the event on the window backbone represents time, *i.e.,* time difference between the timestamp of the event and the open time of the window. The color identifies the type of the event: open (blue), activate (green), resize/move (yellow), collapse/expand (orange), and close (red). When the window backbone is visible, *i.e.,* not covered by boxes representing events, it means that the window is currently open but not in focus. In addition to window events, on each window, the visualization shows a red dot when a source code edit event happens. This visual clue helps to identify which windows were used to perform source code changes.

The visualization of Figure 9.1.a depicts 4 tracks of windows. The main window of tracks 1, 3, and 4 remain open after the session end, *i.e.,* there is no close event and the window backbone continues until the session end. On track 1 there are 2 small-windows, on the second track there is only one. The remaining tracks only have main windows. Towards the end of track 1 we can see that the main window gets minimized and remains collapsed for the rest of the session, *i.e.,* light orange window backbone. On track 2, magnified in the figure, the main window is opened, then active for some time and resized (or moved). Afterwards, it remains active for another time interval and is then quickly resized or moved again. Then an edit event happens on this window, *i.e.,* red dot. After some time the window is resized again and then a small-window is opened that remains open for a little time before giving control back to the main window. After this time, the main window remains active until the focus is given to another window (*i.e.,* the main window of track 1). When the focus is back to this track another edit event happens and after some time the window, and the track, terminate.

Activity Timeline

The *Activity Timeline* (Figure 9.1.b) portrays development activities such as navigation, inspection, edit, and understanding. Using this timeline one can have a clue of when and for how long the developer performed different kinds of activities. Each activity is a box with fixed height. Its length is proportional to the duration of the activity. Navigation activities are white ticks, since the navigation per se lasts for a short amount of time, *e.g.,* 1 second. Other types of events instead have a duration that we estimated from the interaction histories. The color identifies the type of activity: navigation (white), inspection (blue), editing (red), and understanding (yellow).

Vertical Lines: Pause Times and Commits

The last visual elements on the visualization are vertical lines that span both visualizations. There are four types of such lines. Two gray lines without labels indicate the start and the end time of the session. Blue lines without label indicate the timestamp of commits in the version control system. Gray and red lines with label depict pause times. The label of these lines represents the duration of the pause. Gray lines depict the *"explicit"* pause time, *i.e.,* when the developer paused DFlow. Red lines identifies *"implicit"* pause time, or *"idle"* time.

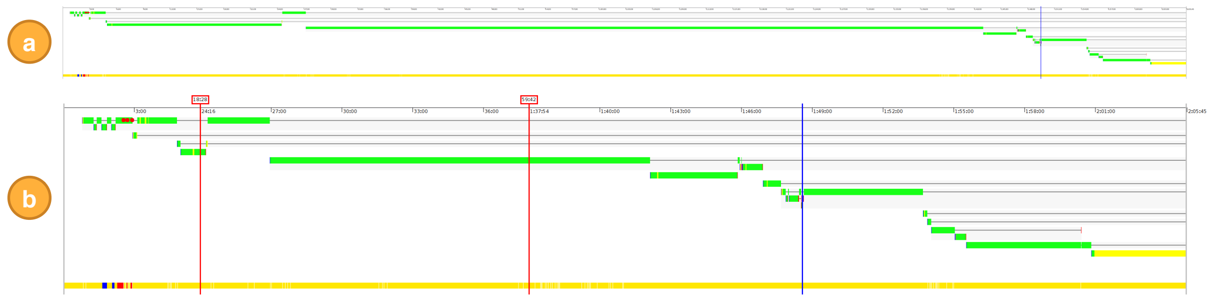


Figure 9.2. Visualizing the same session (a) before and (b) after the removal of pauses

Summarizing the duration of the pauses with vertical lines is essential to have a more understandable visualization. Figure 9.2 depicts same development session visualized (a) including idle time and (b) after idle time removal. As desirable, Figure 9.2.b is more insightful than Figure 9.2.a while maintaining the same pieces of information.

9.2 Telling Development Stories with the UI View

The aim of our visualization is to understand how developers use the UI of the *Pharo* IDE while performing their tasks. In this section we provide details on the dataset for this study and discuss 4 development stories about specific sessions in our corpus.

9.2.1 The Dataset

When we performed this study, DFLOW featured the UI to explicitly pause and resume sessions (please refer to Section 5.3 for additional information about the evolution of DFLOW). The dataset, summarized in Table 9.1, counts around 170 development sessions, totaling more than 110,000 development events and 80,000 interactions on windows. The Table includes average pause time (*i.e.,* the average interval between two explicit sub-sessions) and idle time per session.

Table 9.1. Dataset – Sessions statistics grouped by developer

Dev.	# Sessions	Sub-sessions Total	Sub-sessions Avg.	Avg. Session Duration [hh:mm:ss]	Avg. Pause Time [hh:mm:ss]	Avg. Idle Time [hh:mm:ss]
D1	12	73	6.08	3:01:24	0:01:26	28:42:55
D2	3	3	1.00	0:16:27	0:00:00	00:00:00
D3	65	97	1.49	0:52:32	0:18:12	00:29:20
D4	6	11	1.83	0:48:13	0:18:26	00:58:43
D5	72	202	2.81	0:54:56	3:42:45	00:15:52
D6	7	30	4.29	1:25:18	2:11:36	01:05:37
D7	12	80	6.67	1:34:25	1:41:18	17:29:52
All	**177**	**496**	**3.45**	**1:16:11**	**1:10:32**	**07:00:20**

Table 9.2 outlines statistics about development events, and Table 9.3 outlines statistics about window events. In particular, window events suggest that developers may exploit UI in different ways: For example, D4, D6, and D7 use on average a reduced number of windows. Our visualization aims to highlight further insights on how development sessions are structured.

Table 9.2. Dataset – Development events grouped by developer

Dev.	Navigation	Inspect	Edit	Total
D1	21,617	183	2,458	24,258
D2	393	157	24	574
D3	20,468	2,157	2,091	24,716
D4	2,183	353	1,196	3,732
D5	35,801	2,962	3,316	42,079
D6	6,862	337	472	7,671
D7	7,234	486	526	8,246
All	**94,558**	**6,635**	**10,083**	**111,276**

Table 9.3. Dataset – Window information grouped by developer

Dev.	Windows	Open	Activation	Resize & Move	Collapse & Expand	Close	Total
D1	3,144	2,255	2,488	4,114	143	3,711	12,711
D2	71	63	59	275	0	51	448
D3	3,183	2,518	2,355	4,841	52	2,807	12,573
D4	608	609	134	978	5	710	2,436
D5	7,365	6,088	4,792	28,805	175	7,211	47,071
D6	555	525	549	468	0	580	2,122
D7	769	773	392	3,512	3	691	5,371
All	**15,695**	**12,831**	**10,769**	**42,993**	**378**	**15,761**	**82,732**

9.2.2 Development Stories

Our analysis uncovered a number of interesting stories about how developers work and how they use the UI of the *Pharo* IDE. In this section we report 4 of them.[1]

- The Inspection Valley;
- Implement First, Verify Later;
- Home Sweet Home, and
- Curing the Window Plague.

[1] To increase the readability of this Section, we present each pattern on a new page.

The Inspection Valley

Figure 9.3 shows part of a session recorded by developer D5. The session, excluding pauses, lasted for 49 minutes and 18 seconds.

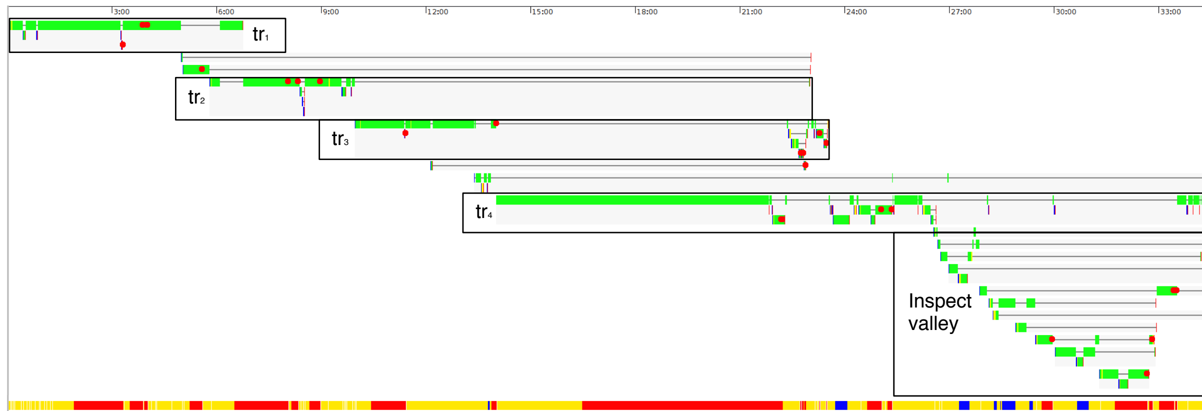


Figure 9.3. Development story for D5: "The Inspection Valley"

The Figure shows the first sub-session lasting 34 minutes. There are 4 subsequent main tracks of windows, denoted as tr_1-tr_4 in the Figure. The session started with tr_1 where the user mainly performed navigation, understanding, and some edits (see the *Activity Timeline*). Then she moved to a new track and performed additional edit operations while triggering a number of small-windows for navigation purposes. The interesting part starts when she moved to tr_4. At the beginning waters are calm: The developer remained for about 8 minutes on the main window of the track. Then she performed a short but convoluted sequence of activities on tr_3 spawning more than 5 small-windows and then she happened to get lost in the *"Inspection Valley"*. Starting from minute 27, in fact, she abandoned all the main tracks to drill down in a series of inspection and understanding activities that led to a series of edit events on different windows. Finally, 6 minutes later, she cleaned up the IDE (*i.e.,* closing most of the windows) and terminates her trip into the inspection valley. These drill downs in "valleys" are a recurrent pattern in a number of sessions. We believe that this practice is encouraged by the multi-window nature of the *Pharo* IDE. It remains to be investigated if this pattern can lead to confusion and whether developers prefer alternative means to perform, for example, chains of inspections on object instances.

Implement First, Verify Later

Developers are often in a rush so they first jump here and there to understand where and what to modify and then start performing changes. Then they run their code, encounter some problems, and spend considerable amounts of time with debugging activities.

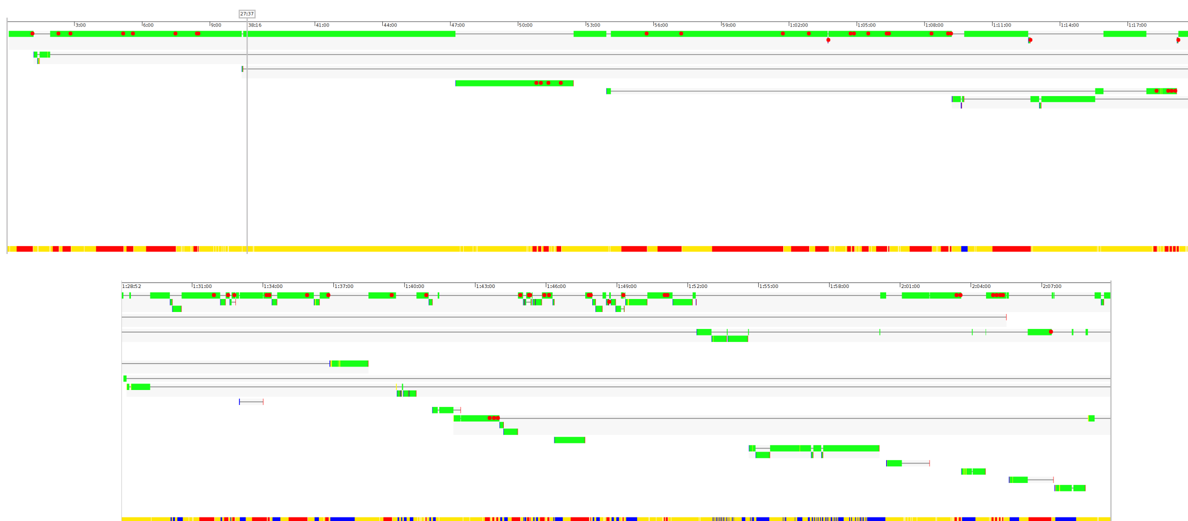


Figure 9.4. Development story for D3: “Implement First, Verify Later”

Figure 9.4 shows a session of developer D3.[2] This session lasts for more than one hour and a half, removing pause times. The session counts three explicit sub-sessions, *i.e.,* vertical gray bars in the visualization, and involves 57 windows. The Activity Timeline reveals two distinct development phases. In the first two sub-sessions the developer mainly acquires knowledge of the system, through navigations (white) and understanding phases (yellow), and performs a number of source code modifications (red). After this, the developer took a break of 8 minutes (*i.e.,* the idle time between the second and third sub-session) and then started to exercise her code. From the Activity Timeline we can infer that she was not really satisfied with her changes. The third sub-session, in fact, is full of inspections (blue) that are often related to debugging activities. In Smalltalk, developers use inspections to observe instances of objects at runtime mostly to verify the values of their fields. In this sub-session inspections are interleaved with a high number of edit activities (red), symptom of the fact that the changes performed in the first two sub-sessions were not really successful.

[2]Figure 9.4 is divided in two parts to better fit the page layout.

Home Sweet Home

The story is about a session of developer D3, depicted in Figure 9.5. The session lasts for about an hour and features 45 edit events. The visualization exposes a single main track of windows, *i.e.,* the first one. This track starts at the beginning of the session, lasts for almost its entire duration and it is intensively occupied by the main window. When this main window is not active (*i.e.,* the window backbone is visible) the developer transfers the focus for a small amount of time to other windows (*i.e.,* maximum 1.5 minutes). She wiggles around, navigates code, reads code, and finally she gets back to the main track. The interesting pattern is that she only performs changes on the first track, more precisely, on its main window (*i.e.,* all but one red dots are in the main track). It is clear that this window is a "pillar" for this session since the flow of development always returns to it.

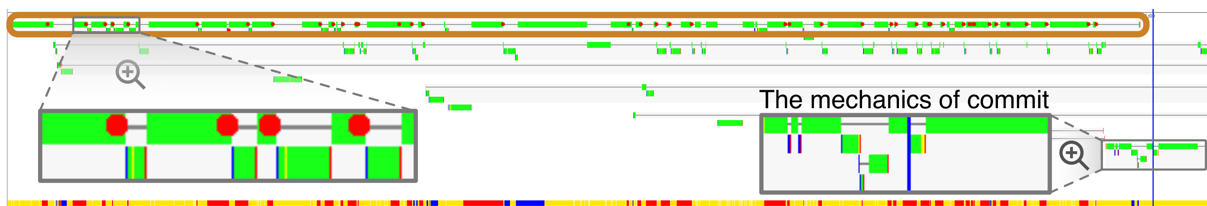


Figure 9.5. Development story for D3: "Home Sweet Home"

Another peculiarity of this session is that, almost after every edit event, the developer opens a small-window, closes it and get back to the main track. It seems that the developer uses small windows as verification means for her changes. An example of this fact is magnified on the left part of Figure 9.5.

The last interesting thing, common to several sessions of different developers, is the visual cluster appearing towards the end of the session, near the blue commit line. This last track of windows, in fact, represents the *"mechanics of commit"*, *i.e.,* the sequence of user interface actions needed to commit source code to the versioning system. The main window of that track is the browser for MONTICELLO, *i.e.,* the version control system used by the *Pharo* IDE. The small-windows originated from it are: i) the *change browser*, that lets the user browse for the changes before commit; ii) the *commit message box*, that lets the user enter a commit message for the current version; and iii) the *confirmation dialogs* that acknowledge every commit.

Curing the Window Plague

Figure 9.6 shows a distinctive feature of developer D5. Researchers called *window plague* the fact that to reveal relationships between code entities developers are forced to open a high number of windows on different artifacts [RND09].

Figure 9.6. Development story for D5: "Curing the Window Plague"

The *window plague* leads to a crowded workspace with many opened windows (or tabs in case of tab-based IDEs). Naïvely the IDE does not come to the rescue of the developer in case of a crowded workspace. Some developers ignore this issue but others like to cleanup their environment from time to time. Developer D5, for example, is a developer that systematically cleans up her environment. Figure 9.6 depicts one of her sessions lasting for more than 4 hours (*note that the view has been compressed to fit the page by eliding parts of the session from the visualization*). At regular intervals (about every hour and a half) she triggers "cleaning stages" where she closes almost all windows to refresh her environment. The UI View shows that the number of windows continuously grow until the entropy level of the environment becomes unbearable for her and she decides to decrease it. There are two possible interpretations for this: Either the developer has accomplished her task and starts a new fresh task, or the environment has become so convoluted and disordered that she decides to start over even though the task is not finished. It remains to be investigated how frequent this phenomenon is and if and how the IDE can automatically come to the aid of developers in such cases.

9.3 Categorizing Developers and Development Sessions

We use this visualization to derive a classification of the development sessions. The ground element for our classification is the number and behavior of tracks of windows. We discarded 13 sessions that were either too short or lacked a significant number of events. We use two dimensions for the classification: i) the presence of dominant tracks of windows, and ii) the flow between the different tracks (*i.e., track flow*).

Dominant Tracks

We define a dominant track as a track with a privileged role in the development session. In other words, dominant tracks are the tracks with a predominant focus time and concentration of edit events. We devise three categories based on *Dominant Tracks*:

- **Single-Track:** There is a only a single track of windows that is predominant over all the others, if any.
- **Multi-Track:** There are two or more tracks of windows that are predominant over all the others, if any.
- **Fragmented:** There are no real dominant tracks of windows. The development flow and the focus of development are strongly fragmented, *i.e.,* the developer continuously shifts her focus from one window to another and perform edits with no apparent strategy.

Track Flow

Track flow describes the way the developer alternates from different window tracks. We devise two additional categories based on *Track Flow*:

- **Sequential Flow (S):** The development flow follows a sequential trend, *i.e.,* from one track the focus moves to the next and so on. On this type of sessions the focus rarely goes back to a previous track. These are the sessions that might suffer from the *window plague* [RND09], and in such cases often there are no dominant windows.
- **Ping-Pong Flow (PP):** The development flow and the focus of development continue to *zig-zag* between two or more tracks. If the fragmentation is heavy, there can be no dominant track and the visualized session appears frenetic and chaotic.

Consider the developer stories we described in Section 9.2.2. Figure 9.3 is mostly fragmented since there are no dominant windows that characterize the whole development session. It also has a dominant sequential flow, because the developer starts a development track, and when another one is created the developer has either closed the previous one or leaves it out of focus. There is also minimal ping-pong behavior.

The session in Figure 9.4 is single-track in the implementation phase, while in the verification phase it becomes fragmented with a mixed sequential and ping-pong flow.

Consider instead Figure 9.5, corresponding to the “Home Sweet Home” developer story. The session has a main track where most of the edits happen, and that the rest of development activities happen in other window tracks. The session has a typical ping-pong flow. In fact, the developer frequently alternates between the main track and other tracks, with a minimal privilege towards the second track.

Finally, the session in Figure 9.6 shows instead a fragmented session with sequential flow. Edits and focus are spread to many tracks, and no track dominates in the whole session. A lot of old tracks are simply out of focus and never closed, laying in the background (*window plague* [RND09]). Tracks grow almost monotonically.

Table 9.4. Results – Track and Flow characterization of developer session

	Single-Track				**Multi-Track**				**Fragmented**				**Not Classified**		
	S		**PP**		**S**		**PP**		**S**		**PP**				
Dev.	#	%	#	%	#	%	#	%	#	%	#	%	#	%	**Total**
D1	0	0%	1	8.3%	1	8.3%	0	0%	8	66.7%	0	0%	2	16.7%	12
D2	2	66.7%	0	0%	0	0%	0	0%	1	33.3%	0	0%	0	0%	3
D3	32	49.2%	13	20.0%	6	9.2%	4	6.2%	7	10.8%	1	1.5%	2	3.1%	65
D4	0	0%	0	0%	0	0%	0	0%	5	83.3%	1	16.7%	0	0%	6
D5	10	13.7%	3	4.1%	9	12.3%	1	1.4%	41	56.2%	0	0%	9	12.3%	71
D6	1	14.3%	1	14.3%	1	14.3%	1	14.3%	2	28.6%	1	14.3%	0	0%	7
D7	5	45.5%	1	9.1%	1	9.1%	1	9.1%	3	27.3%	0	0%	0	0%	11
All	**50**	28.2%	**19**	10.7%	**18**	10.2%	**7**	4.0%	**67**	37.9%	**3**	1.7%	**13**	7.3%	**177**

Table 9.4 illustrates the characterization of our corpus of development sessions, aggregated by developer. In general, we observe that Multiple-Tracks are less common (14.2% of the total), and that the remaining sessions are uniformly distributed among the other two categories: Single-Track and Fragmented (around 40% of each type). We also observe that Sequential Flow (S) is relatively more frequent than Ping-Pong (PP), 76.3% versus 16.4%. These numbers again support the fact that developers may frequently experience the *window plague*. Another interesting general observation is that Ping-Pong behavior is mostly correlated with Single-Track and Multi-Track sessions: This means that Ping-Pong Flow happens between a single dominant track and minor tracks, or between multiple dominant tracks.

The results of our classification also suggest differences between the development style of different developers. Three out of seven developers (D2, D3 and D7) exhibit a strongly dominant preference for Single-Track sessions: For them such sessions account for more than 50% of the total. Developers D1 and D4, instead, tend to work in a more Fragmented fashion (66.7% and 83.3% respectively). All developers strongly prefer Sequential Flows; this result can be debated for developer D6, for which we did not collect enough sessions to observe a significant difference. Developer D6 is the subject that more frequently exhibits Ping-Pong Flow in her sessions (42.9%). Developer D3 is the subject with the higher number of sessions with Ping-Pong Flow but, in percentage, has a smaller number than D6 (27.7%).

Considering developer styles in isolation, we observe that sessions of developer D5 are almost only Sequential (82.2% of the sessions) and frequently Fragmented (56.2%). Instead, sessions of developer D3 are still mostly Sequential (69.2% of the times) but mostly exhibit a Single-Track of development (69.2% of the sessions).

9.4 Reflections

This chapter presented a visual approach to understand how developers use the UI of an IDE starting from raw interaction histories collected with DFlow. Together with the use of the UI of the IDE, the view also provide insights about when and how developers perform different development activities such as navigating, writing, and understanding source code, *i.e.,* activity timeline. Our visualization enables the identification of main development tracks in terms of

usage of IDE UI components like windows, and relates such usages with development activities like edit events, inspection events, and commits. To gather data we used DFlow, our tool that silently records all fine-grained interactions with the IDE.

We discussed four developer stories from the visualized sessions, that identified both peculiar developer behaviors emerging from the usage of the IDE and their activities, and well known phenomena like the *window plague* [RND09]. We also proposed a simple classification of the visual features of development sessions in terms of dominant window tracks (Single-Track, Multi-Track, and Fragmented sessions) and Flow between tracks (Sequential or Ping-Pong). By using such a classification, we found that different developers exhibit different behaviors and usages of the UI of the IDE. For example, most developers either work with a Single or no dominant window track, with a strong prevalence of Sequential Flows that may lead to cluttered environments.

10

Visualizing the Evolution of Working Sets

While the interaction with the IDEs apparently has the sole, ultimate effect of creating and modifying source code, it also generates myriads of events that capture the various mechanisms of actual development. This data reflects the *modus operandi* of a developer, including the UIs and IDE components she uses more frequently [MMRL16] and enumerates all the program entities she interacted with during a session.

According to Wexelblat and Maes, the information path obtained from navigating in an information space exposes and reveals the mental model of the system as perceived by a user [WM99]. In the case of software development, the set of entities navigated and interacted with, compose the "*working set*" (also called *context*) that developers leverage to create and maintain their mental model of the software system at hand supporting their current development task.

Maintaining the working set is an essential part of program comprehension that absorbs a considerable portion of development time [Cor89]. However, this process is often inefficient and not properly supported by IDEs. Many studies have shown evidence of issues related to navigation and the maintenance of working sets. For example Fritz *et al.* discovered that, on average, the context model necessary to solve a task is composed of 4 classes and a subset of their methods [FBM+14]. Ko *et al.* found that 27% of the navigations concern visits to program entities that have been already visited [KMCA06]. The study also observed interesting patterns of *back-and-forth navigation* to compare related pieces of code. In Chapter 7 we discussed the issues developers have while navigating source code in an IDE. We believe this is a manifestation of the problem of maintaining working sets [MML15b]. In Chapter 8 we modeled and empirically measured the actual navigation efficiency of developers compared to different ideal scenarios, finding that there is significant room for improvement [MML16a].

In this chapter, we present a visualization to characterize how working sets evolve during a development session. The visualization depicts the intensity of the developer activity on entities of the working sets, and the navigation paths that occur between them. We visualized 914 development sessions coming from 14 developers and identified visual patterns on the evolution of working sets during development.

Structure of the Chapter

Section 10.1 defines what is a working set and illustrates the principles of our visualization. Section 10.2 presents a catalog of patterns emerged from a visual analyses of development sessions. Finally, Section 10.3 concludes the chapter.

10.1 Visualizing the Working Set

We propose a visual approach to understand how working sets evolve during development sessions. Figure 10.1 shows our visualization in a nutshell.

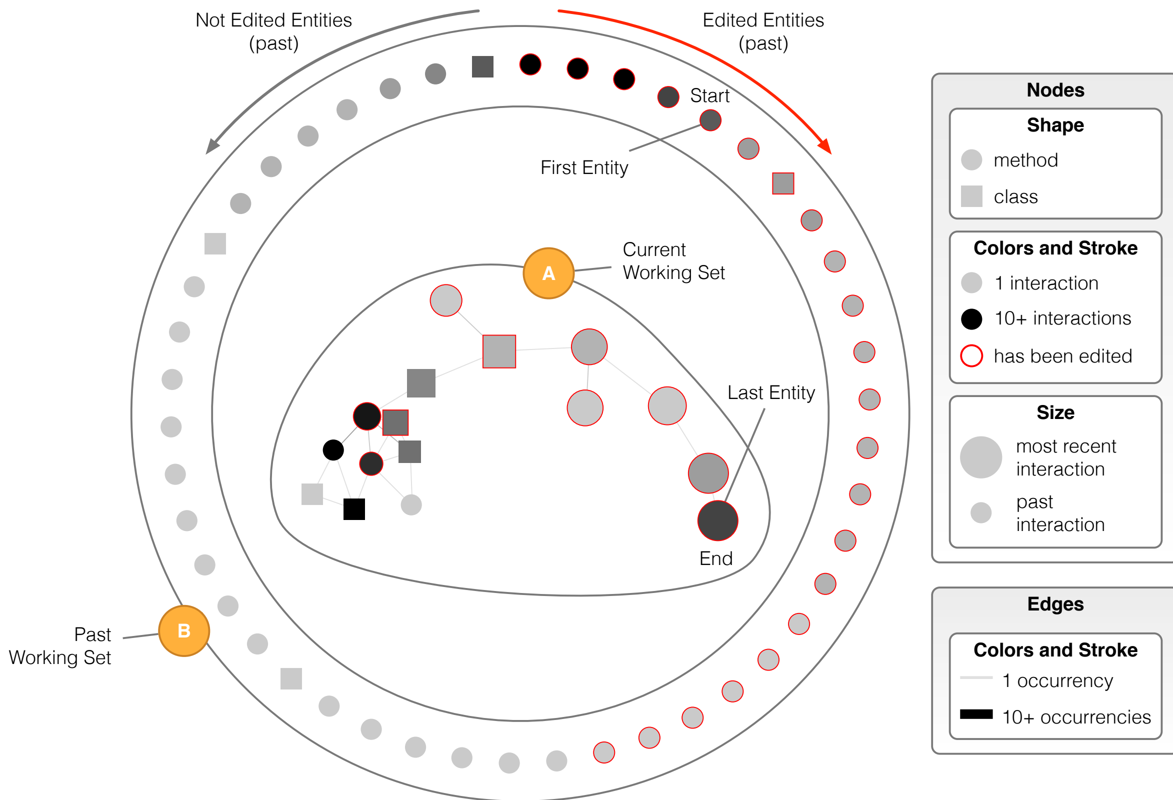


Figure 10.1. Visualizing working sets: visualization principles

10.1.1 What is a Working Set?

While exploring software systems, developers visit and modify different program entities such as classes as methods. We call *"working set"* the group of program entities which a developer has interacted with during a particular period of time. To identify the working set we observe the meta-events recorded with DFLOW: i) *navigation events*, *e.g.,* opening a class definition; ii) *edit events*, *e.g.,* modifying a method's source code; iii) *inspection events*, *e.g.,* checking the state of an object at runtime; we consider its class as the interacted entity. Our definition of working set is similar to what Kersten and Murphy call *"task context"* [KM05]. MYLYN is one of the seminal approaches aimed at improving the construction and management of working sets. The tool leverages developers' interaction with the IDE to build a degree-of-interest model (DOI), filtering the views of the *Eclipse* IDE from entities with a low DOI value [KM05, KM06]. Other tools, such as NAVTRACKS [SES05] and TEAMTRACKS [DCR05], monitor IDE interactions to help developer to navigate the software space, thus supporting the construction and management of working sets. NAVTRACKS supports this task by visualizing related program entities with a simple graph-based view.

10.1.2 Visualization Principles: Nodes, Edges, and Layout

Figure 10.1 shows a development session depicted using our visualization. The view is made of two parts, depicting the current (Fig 10.1.A) and the past (Fig 10.1.B) working set respectively.

Current Working Set. The group of entities visited by the developer in the last timeframe. We define the last timeframe in terms of i) number of interactions (*i.e.,* the last 30 interactions) or ii) in a temporal fashion (*i.e.,* the interactions happened in the last 10 minutes).

Past Working Set. The Past Working Set is composed of all the entities the developer interacted with in the past, *i.e.,* before the *current working set.*

Nodes: Program Entities

In the visualization, nodes represent program entities (*i.e.,* methods and classes) the developer interacted with during a development session. Methods are depicted using circles, while classes are depicted using squares. Each node is colored using a gray-scale denoting the intensity of the interaction on the program entity it represents. A light gray node is a node with one (or a few) interactions. A node depicted in black is a node with 10 or more interactions, *i.e.,* the color scale saturates at 10 to make the visualization more simple to understand.

We distinguish two kinds of interactions: Interactions that do not modify the source code of the entity (*e.g.,* reading a class definition) and interactions that modify it (*e.g.,* editing the body of a method). The visualization adds a red border to the nodes that have been involved in at least one edit operation.

The size of each node (*i.e.,* diameter for circle and side for square) depicts the recency of the interaction on the corresponding entity. By default, all nodes have a standard size of 20 pixels. The last interacted node has double the standard size, and the nodes targeted by the last 10 interactions follow a linear scale from this double size to the standard size.

The view uses the labels *Start* and *End* to denote respectively the first and the last program entity the developer interacted with in the visualized interaction histories.

Edges: Flow of Interactions

Edges express the flow of the interactions, and not structural source code properties. An edge between `method Foo` and `class Baz` means that, in the interaction history, two subsequent events involved these two program entities (*e.g.,* a navigation event from `Foo` to `Baz`). For simplicity, edges are undirected: The edge between `method Foo` and `class Baz` summarizes all the interactions between these two nodes.

Both the color and the stroke width of edges are mapped to the same metric, *i.e.,* the number of times the path represented by the current edge is followed by the developer in the interaction history. Both features are bounded. An edge depicted in light grey represents a path that is traversed one (or a few) times in the interaction history. A path depicted in black represent a path that has been crossed 10 or more times by the developer. The stroke width is bounded between 1 and 20 pixels (*i.e.,* that corresponds to the standard size of nodes). It follows a linear scale between the minimum number of occurrences of the path (*i.e.,* 1) and the maximum occurrences, computed using all the interactions of the entire session.

Layout

To depict current and past working set the view uses two different layouts. For the current working set (Figure 10.1.A) we use a force-directed graph layout. Figure 10.2 depicts the underlying mechanics of the force based layout.

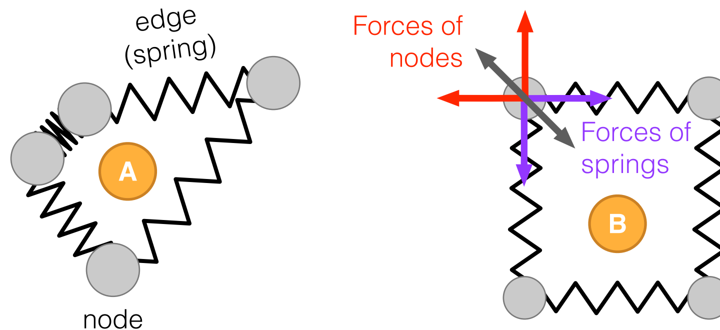


Figure 10.2. The principles of the force-based layout

Each node has a *charge* that tends to repulse other nodes. At the same time, edges act as *springs*, thus tend to assume their "ideal" length, *i.e.,* neither too compressed nor too spread. From an initial random configuration, depicted in Figure 10.2.A, the layout will progressively move the nodes trying to minimize all the forces exercised on the graph: the repulsive forces of nodes (*i.e., charge*) and the attractive forces of edges. Figure 10.2.B shows a configuration where forces are minimal. At the beginning all nodes have the same *charge* and all edges have the same *strength* and *ideal length*,[1] however as further discussed in the next section, these parameters evolve together with the evolution of the working set.

The outer part of the view (Figure 10.1.B) depicts the past working set using a radial (or circular) layout where nodes are equidistant. The radius of the layout depends on the inner part of the visualization, *i.e.,* the bigger is the space occupied by the inner part, the larger will be the radius of the circular layout for the past working set. Nodes are sorted according to the intensity of the interactions. Starting from the top of the circle (*i.e.,* 90°), nodes representing edited entities are placed clockwise, while the ones representing non-edited ones are placed counterclockwise.

Interaction & Customization

The user can interact with the view by panning and zooming. In addition, by hovering on a node, the view shows a tooltip with additional information such as the name of the entity, the number of interactions and, if applicable, the number of edits on the hovered entity. Clicking on a node lets the developer conveniently jump to the respective class/method definition to read (or modify) the code in the code browser of the *Pharo* IDE. The view can be customized by changing the number of interactions considered recent (*i.e.,* impacting the size of nodes) or the criteria to distinguish between past and present working sets (*i.e.,* impacting the overall look of the view and its meaning).

[1]Node charge: -1,000; Edge strength: 1; Edge ideal length: 35.

10.1.3 Co-Evolution of Working Set and Visualization

Our goal is to present an evolutionary view to depict how the past and current working sets evolve during a session. To make this possible, the visualization co-evolves with the working set, as explained in Algorithm 1.

```
   Data: A sequence of events (i.e., InteractionHistory)
1  view ⟵ initialize an empty view
2  lastEntity ⟵ null
3  for event ∈ InteractionHistory do
4  |   currentEntity ⟵ extract the entity from the event
   |   // Adding or updating the node
5  |   if view contains currentEntity then
6  |   |   UpdateNode(currentEntity)
7  |   else
8  |   |   add currentEntity to the view
9  |   end
   |   // Adding or updating the edge
10 |   edge ⟵ edge from lastEntity to currentEntity
11 |   if view contains edge then
12 |   |   UpdateEdge(edge)
13 |   else
14 |   |   add edge to the view
15 |   end
16 |   ApplyLayout(view)
17 |   ApplyAging(view)
18 |   lastEntity ⟵ currentEntity
19 end
```

Algorithm 1: Constructing the View

First, for every event in the interaction history, the algorithm checks whether the program entity is already visualized and updates it, otherwise it adds it to the view.

Second, it applies the layout and the aging mechanism on all the nodes and edges of the visualization, as described in Algorithm 2 and 3 respectively.

```
1 method UpdateNode (node) :
2 |   update color of node (using # interactions)
3 |   reset size of node (to max size, since last visited)
4 |   reset time−to−live (TTL) of node (to default value)
5 |   if node has been edited then
6 |   |   add a red stroke
7 |   end
8 |   charge ⟵ 80% ∗ charge
9 end
```

Algorithm 2: Updating a Node in the View

Every time a node is re-visited, its color is updated with a linear grey-scale representing how many interactions have involved that entity. Its size is restored to the maximum size, since it is the last visited node. The algorithm also resets the time-to-live (TTL) of the *node* to the default value, *i.e.,* 30. The TTL is used to distinguish between current and past working set:

When the TTL reaches 0, the *node* no longer belongs to the current working set. The last line of Algorithm 2 updates the node *charge*, used by the force-directed graph layout to arrange the current working set. At the beginning each node has the same initial negative charge; for how the layout is implemented, negative charges tend nodes to repulse themselves. For each interaction with an entity, we decrement its node charge by 20%, making the node less repulsive, and obtaining a more compact view of the current working set.

Algorithm 3 explains how we update each edge.

```
1  method UpdateEdge (edge) :
2  |   update color of edge (using # interactions)
3  |   update width of edge (using # interactions)
4  |   strength ⟵ 120% ∗ strength
5  end
```

Algorithm 3: Updating an Edge in the View

An edge color is mapped to a linear grey-scale that represents how many times the edge has been walked by the developer. The same information is also encoded in its stroke width. The last step is the update of the edge *strength*, used by the force-directed graph layout for the current working set. The strength represents the force of attraction for the edges. A high value results in having nodes together. At the beginning each edge has the same initial strength (*i.e.,* 1). Every time that an edge is exercised, we increment its strength by 20%, bringing the two connected nodes closer.

Our approach considers the recency of the last interaction on a node as a key factor to determine the current and the past working set, as explained in Algorithm 4.

```
1  method ApplyAging (view) :
2  |   for node ∈ view do
3  |   |   decrease the time − to − live (TTL) of node by 1
4  |   |   if size of node > minNodeSize then
5  |   |   |   decrease size of node
6  |   |   end
   |   |   // Disconnect node if TTL elapsed
7  |   |   if TTL of node = 0 then
8  |   |   |   disconnect the node from the graph
9  |   |   end
10 |   end
11 end
```

Algorithm 4: Applying the Aging mechanism to the View

After each interaction event, we iterate over all nodes and we decrement their time-to-live (TTL). Since the size of each node corresponds to the recency of the last interaction on the node itself, as nodes become old, we reduce their size (if its size is not already below the *minNodeSize*, *i.e.,* 20 pixels). The last part of the method checks whether the TTL of a node is elapsed and disconnects it from the graph. If the node has not been targeted by any interaction in the last 30 iterations of Algorithm 1, it leaves the current working set.

Finally, the algorithm applies the force-directed graph layout for the current working set and radial layout for the past working set. Algorithm 5 illustrates this process.

After the aging process described in Algorithm 4, our approach can identify the current and the past working set. The current working set is composed of all the nodes that are connected in the visualization. To these nodes, we apply a force-directed graph layout, using the up-to-date

```
1 method ApplyLayout (view) :
2     currentWS ⟵ connected nodes in the view
3     pastWS ⟵ disconnected nodes in the view
4     sortedPastWS ⟵ sort pastWS according to number of interactions and edits;
5     apply force-based layout to currentWS
6     apply radial layout to sortedPastWS
7 end
```

Algorithm 5: Applying the two layouts to the View

$charges$ and $strengths$ computed in Algorithms 2 and 3 respectively. Among the advantages of this layout, the obtained visualization is aesthetically pleasing, simple, and intuitive.

The remaining, disconnected nodes represent the past working set. We layout the past working set with an equidistant radial (or circular) layout. We set its radius so that the representation of the *currentWS* fits. The circular layout ensures that all the nodes are treated neutrally, since they are at equal distances from each other and from the center of the visualization [INM+05]. In addition, in our layout nodes are sorted according to the number of interactions and the editing status, *i.e.,* whether the represented program entity has beed edited in the past or not. This does not affect the neutrality of nodes, but enables a quicker assessment of which entities from the past working set have been interacted (or edited) the most.

10.2 Visual Analysis: Dataset and Patterns

We visualized the evolution of a large set of development sessions collected with DFlow. The analysis revealed a number of patterns referring to a single snapshot (Section 10.2.2) and evolutionary patterns (Section 10.2.3).

10.2.1 Dataset

We applied our visualization to 914 development sessions, collected with DFlow, coming from 14 developers (open-source developers and PhDs). As in all other studies, developers were not given a task, but rather they were recorded while performing their daily activities. Table 10.1 summarizes the dataset used for this study.

Table 10.1. Dataset – Totals values and values aggregated per session

All	Total
# Sessions	914
# Developers	14
# Snapshots	72,631

Per Session	Avg.	Q_1	Q_2	Q_3
# Snapshots	79.21	18	35	87
Working Set (WS)	9.57	4.70	7.13	12.10
Past WS	2.98	0.00	0.33	3.23
Current WS	6.59	4.00	5.86	8.78
Connectedness (%)	21.59%	15.67%	20.94%	26.78%

Our visualization of working sets in development sessions is evolutionary and incremental. Thus, for every session, we identify a number of *snapshots* to build a step of the visualization.

We define a snapshot as a moment in time in which a program entity is either visited for the first time, re-visited, or modified. In total we identified 72,631 snapshots in our entire dataset.

The lower part of Table 10.1 reports the snapshot data aggregated per session. On average, each session has 79.21 snapshots (with a median of 35). The working set (WS), on average, is composed of 9.57 entities (on average 2.98 entities form the past working set and 6.59 the current working set). The last metric we report is the percentage of *connectedness*.
Given an undirected graph with n nodes, the maximum number of edges ($edges_{max}$) is:

$$edges_{max} = \frac{n \cdot (n-1)}{2}$$

Considering this as an upper bound, we can measure the percentage of *connectedness* of a graph with a given number of edges ($|edges|$) as:

$$connectedness\ (\%) = \frac{|edges|}{edges_{max}}$$

We compute the connectedness of the current working set. The connectedness expresses the average probability of two entities to belong to at least 1 subsequent pair of events in the recency window defining the current working set. On average our current working sets graphs have a percentage of connectedness of 21.59% (and a median of 20.94%). The most connected working set, not shown in the table, has a percentage of connectedness of 50%.

Visual Patterns. The following two Sections (10.2.2 and 10.2.3) lists the visual patterns emerged from the visual analysis of development sessions.[2]

Snapshot Patterns

- U Can't Touch This
- Past: To Edit or Not To Edit
- The Guiding Star
- Stay Focused, Stay Foolish!
- Moving in Circles

Evolutionary Patterns

- The Past Awakens
- Multi-Part Session
- Thirst for Knowledge
- The Working Funnel

[2]To increase the readability of this Section, we present each pattern on a new page.

10.2.2 Snapshot Patterns

As the name suggests, snapshot patterns emerge by visually inspecting single snapshots of a development session, *i.e.,* a particular state of the past and current working sets of sessions. The remainder of this section discusses the 5 most interesting patterns that we found.

U Can't Touch This

There are snapshots which are entirely exploratory, *i.e.,* they lack edits both in the past and current working sets. Figure 10.3 shows an example of this pattern.

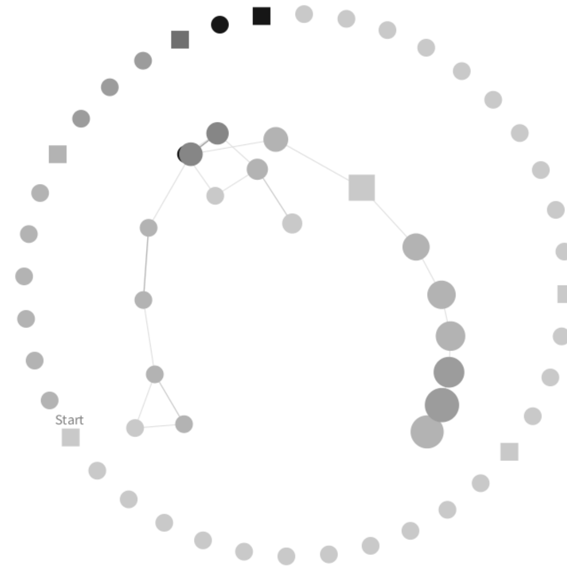


Figure 10.3. Example of the "U Can't Touch This" pattern

Across the entire history, this session has, on average, a working set composed of 39 entities (22 in the past and 17 in the current). The figure depicts the 120th snapshot (out of 147) of the session. Only in the last few snapshots the developer edits 3 program entities.

Potentially, sessions manifesting this pattern are sessions in which the developer is addressing a complex task that requires a very deep understanding of the system, that is consistent, for example, with a complex debugging activity. After a deep phase of exploration, the developer acquired the knowledge to perform few localized changes.

Past: To Edit or Not To Edit

In our analysis we identified a number of session snapshots in which the past working set has a remarkable size but it contains no edit events (*i.e.,* no node in the past working set has a red stroke). Figure 10.4a depicts an example with 52 non-edited entities in the past working set.

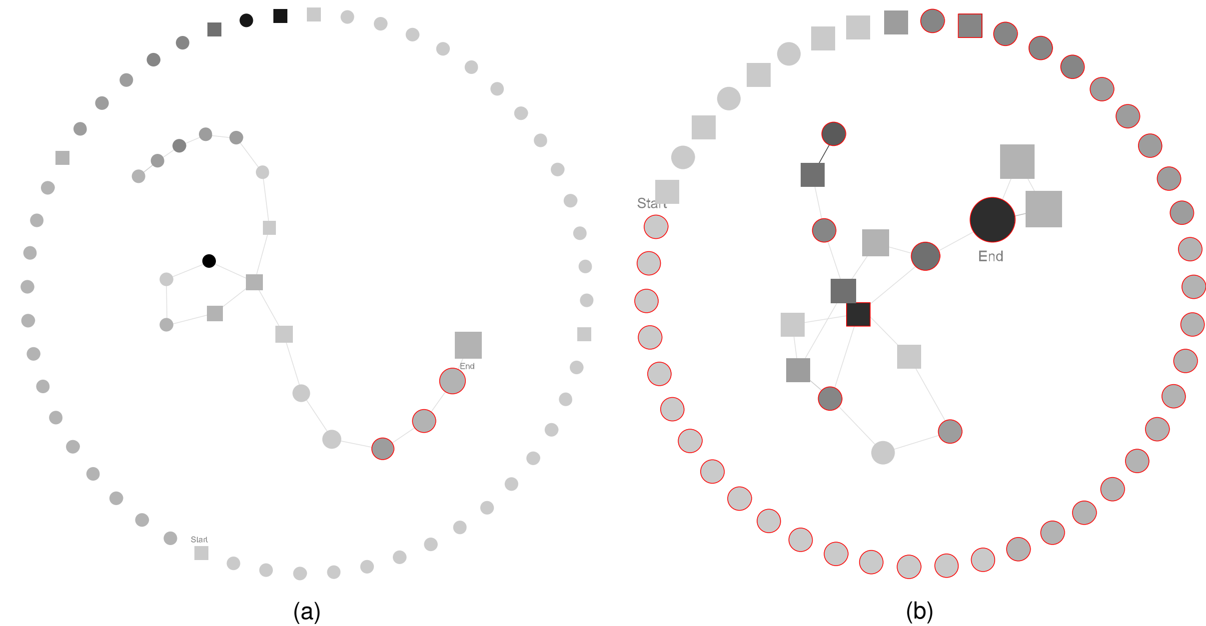


Figure 10.4. Two examples of the "Past: To Edit or Not To Edit" pattern

This means that all the events performed in the past were explorative, targeted at the navigation of the system at hand. We conjecture that the developer needs to build her mental model prior to start her task, consistent with a significant time spent on program comprehension [Cor89]. On the other hand, there are snapshots in which the past working set counts an high number of edits, as in Figure 10.4b. In this snapshot more than 75% of the entities composing the past working set have been edited. Essentially, this could mean that the developer completed a given task and moved to a new one on separate entities.

The Guiding Star

A development session potentially involves a large number of program entities. However, during development there might be a few landmarks that the developer uses as *guiding stars* for her exploration process. Figure 10.5 shows snapshot of a development session that clearly manifests this pattern.

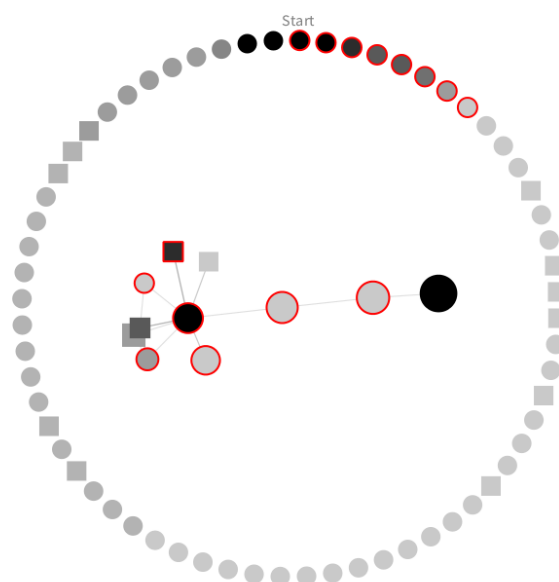


Figure 10.5. Example of the "The Guiding Star" pattern

On the left side of the current working set we can see a circle depicting a method colored in black with 7 connected edges. This method likely plays a key role in the development session, or better in the current working set, supporting the exploration of 8 other entities, *i.e.,* 4 methods and 4 classes.

Another observation is that the number of entities colored in black is relatively low. Since the color represents the number of interactions, this is consistent with the fact that developers need to periodically revisit some key entities (as observed by Ko *et al.* [KMCA06] and Soh *et al.* [SKG+13]), but also with the fact that the context model necessary to solve the task is often relatively small, *i.e.,* 4 classes (as observed by Fritz *et al.* [FBM+14]). Moreover, the edges are relatively long, even the ones connected to the guiding star. This means that the cognitive jumps between the guiding star and a given connected entity are relatively few (since the edges are thin) and equally distributed among the related entities. This could be consistent, for example, with a small, limited refactoring.

Stay Focused, Stay Foolish!

Some sessions have a pattern similar to the guiding star, but involving a greater number of entities that are highly interacted with between themselves. In other words, the snapshot has a sort of "guiding constellation", where the current working set is highly focused on a set of entities, instead of a single one like the case of the guiding star. We call a working set focused if there is a subset of entities that are tightly connected between themselves and have a dark color, symptoms of a high number of interactions. Figure 10.6 shows a snapshot of a session manifesting this pattern.

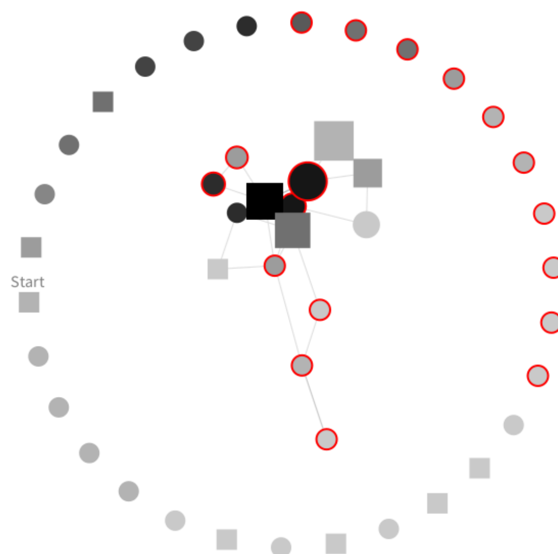


Figure 10.6. Example of the "Stay Focused, Stay Foolish!" pattern

The top part of the current working set is very focused. Some nodes are very dark *i.e.,* they have been involved on a lot of interactions. Furthermore, they are tightly connected, meaning that there have been a lot of cognitive jumps between all the involved entities. Finally, we observe that the last interactions happen on a subset of the nodes in the focus (*i.e.,* some nodes are significantly bigger), meaning that this snapshot belongs to a task which is still revolving around the focused entities.

Moving in Circles

As the name suggests, in the sessions manifesting this pattern developers follow circular paths to explore and eventually modify the software system at hand. Figure 10.7 depicts two snapshots of two different development sessions manifesting this pattern.

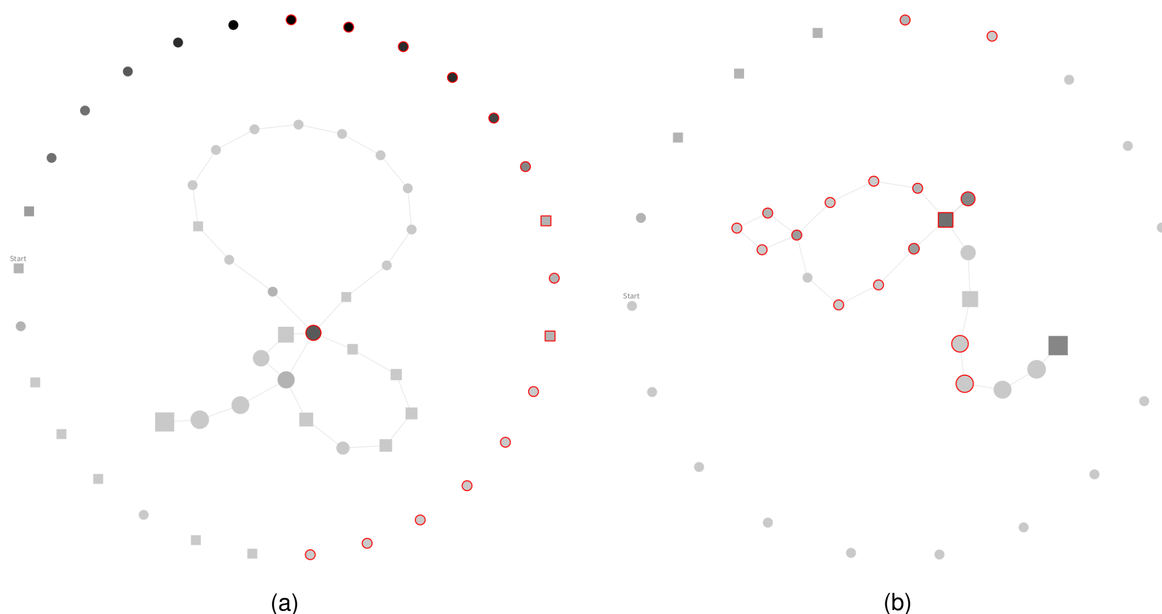


Figure 10.7. Two examples of the “Moving in Circles” pattern

In Figure 10.7a there are two large circular paths, one composed of 8 and the other of 14 entities. It is interesting to notice how all the entities composing the circular paths are only observed by the developer and never modified. The only modified entity in the current working set is the central dark grey entity that apparently acts as a small guiding star for the navigation. We conjecture that circular paths represent side exploration of the system at hand aimed at reinforcing the developer’s mental model before—or during—the execution of a task at hand.

A variation of this pattern sees edited entities inside circular paths, as exemplified in the session depicted in Figure 10.7b. This can be consistent in a manual refactoring involving a sequence of methods of the same class, for example, that does not need to revisit the edited entities (*e.g.*,, in the case of a manual rename of a field).

10.2.3 Evolutionary Patterns

Evolutionary patterns consider multiple subsequent snapshots during the evolution of a development session. In this section we discuss 4 evolutionary patterns that we discovered in our visual analysis of development sessions.

The Past Awakens

During a session, the working set evolves: After taking part in the current working set, entities get old and move to the past working set. However we discovered that there are sessions in which entities also go through the reverse path: From the past (working set) they jump again into the current working set.

Figure 10.8 shows an example of "The Past Awakens". In *Part 1*, all the entities are in the current working set. Then, due to the aging process, in *Part 2*, the past working set grows to 11 entities, accommodating all the entities that the developer is likely not to need in a short time. *Part 3*, instead, exhibits "The Past Awakens": From the 11 entities the past working set shrinks to 9 entities, symptoms that 2 entities have jumped back into the current working set.

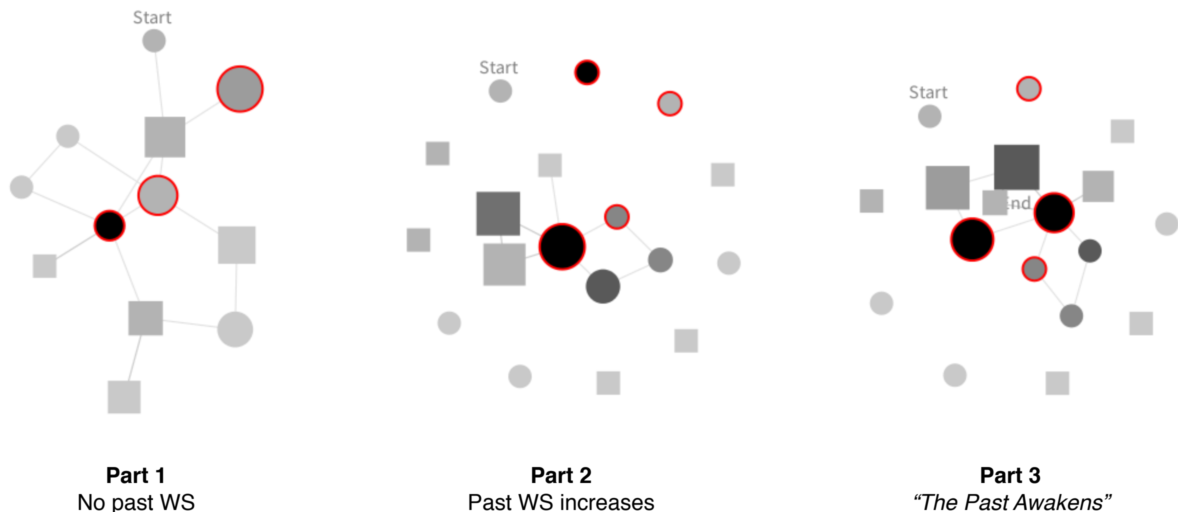


Figure 10.8. Example of the "The Past Awakens" pattern

The manifestation of this pattern adds evidence to the fact that developers need to revisit entities. According to Ko *et al.*, almost one third of the navigations target entities that have been already visited in the past [KMCA06]. We compare our data with their findings by considering each of our snapshots as a form of "navigation". On average, 4.53% of the snapshots of each session manifest this pattern. Even though these preliminary findings seem to contradict Ko *et al.*, the fact that an entity comes back from the past working set is more restrictive than a revisit.

Multi-Part Session

A development session is a sequence of conceptually related events happening in a relatively short timeframe. However, we can often identify clear subsets of events that correspond to precise activities or phases. Examples include source code exploration, debugging, source code modification, etc. We call "Multi-Part Session" a session exhibiting this pattern.

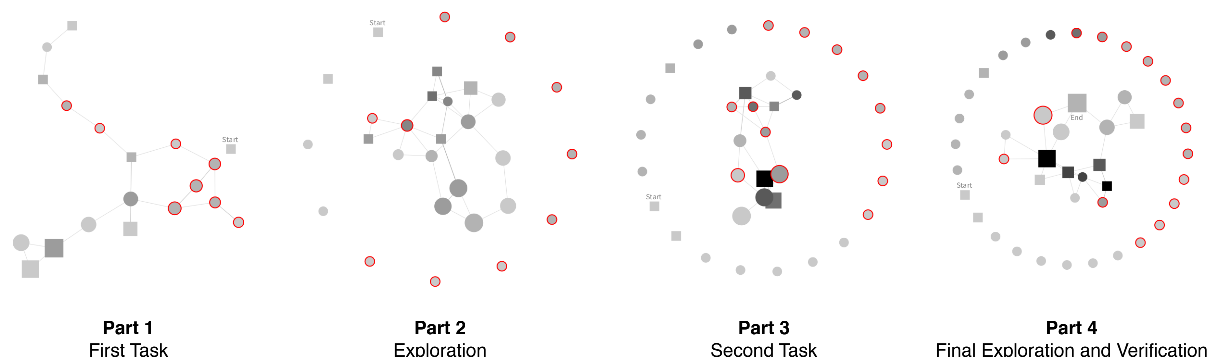


Figure 10.9. Example of the "Multi-Part Session" pattern

Figure 10.9 shows an example of this pattern. In *Part 1* the developer addresses a task: She explores a set of entities and performs edit operations on 8 entities. In this first part, all nodes (except for the *Start* node that alone composes the past working set, *i.e.,* there are no edges connected to it) are in the current working set. In *Part 2*, the developer explores a different part of the system (*i.e.,* the past working set starts populated with 12 entities). She jumps from one entity to the other performing only 2 edit operations, possibly to augment or refine her mental model prior to performing a new task. In *Part 3* the developer edits two new entities and keeps interacting with some of the entities she has navigated in the second explorative part. The last part of the session (*i.e., Part 4* in Figure 10.9), is mostly explorative: All the edited entities go, or remain, in the past working set. Our conjecture is that in this last phase the developer explores the entities related to the ones that she modified during the session to verify the side effects of her modifications.

Our visualization supported us in visually identifying different development activities and interesting snapshots that otherwise would have been non trivial to find.

Thirst for Knowledge

Developer are often confronted with unfamiliar code or code that does not work and need to be fixed. When this happens they need to spend time in performing program comprehension and related activities. As depicted in Figure 10.10, this phenomenon is visible from our visualization that (mostly) portrays entities without the red stroke.

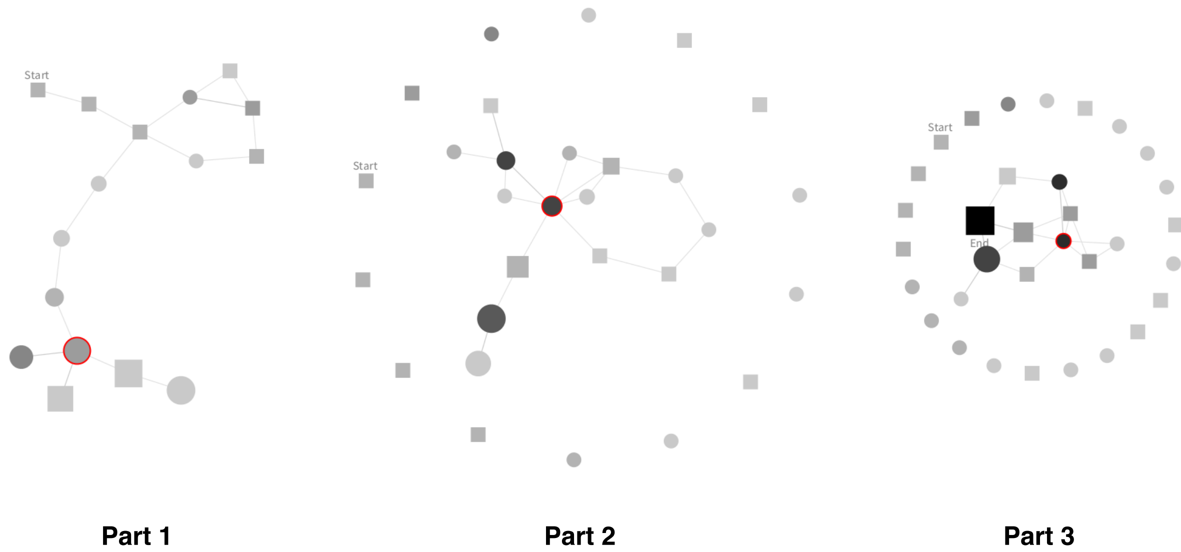


Figure 10.10. Example of the “Thirst for Knowledge” pattern

The Working Funnel

We observed that often the number of program entities that a developer interacts with in the initial phases of a development session is larger than the ones she interacts towards the end of the session. This can be attributed to several different factors. One possible reason for that is the fact that, prior to performing source code changes, developers need to gather a strong knowledge of the system by exploring it. As a result, in the initial parts of a session there are few edit operations but a lot of interactions, symptoms of an exploratory phase.

Figure 10.11 shows 4 snapshots of a development session that exhibits this behavior. In *Part 1* there are no edits, but a chain of explorative events. In *Part 2* the developer starts to modify a handful of entities, while continuing the exploration. In the remaining two parts of the session (*i.e., Parts 3* and *4*), instead, the number of entities in the current working set significantly shrinks. This is the symptom that the developer had stopped exploring. A possible explaination is that she is checking whether her modifications have the desired effects on the entities potentially affected by those changes.

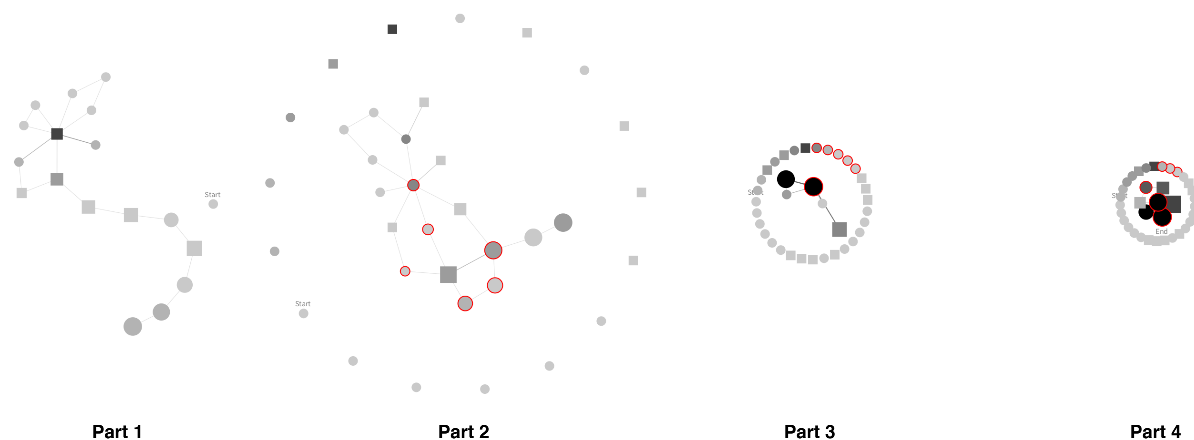


Figure 10.11. Example of the “The Working Funnel” pattern

A developer with a clear mental model of the system or that is facing an easy task, instead, could start by editing a large set of entities right away. Then, in a later part of the session, she could restrict her current working set to a handful of entities to perform the last, non trivial and more focused set of changes that finalizes her task.

Figure 10.12 depicts 5 snapshots of a development session that shows this scenario. In *Parts 1* and *2* the developer explores and modifies 21 entities. *Parts 3* is a steady phase in which the developer explores some entities and performs a few modifications. In the remainder of the session, *Parts 4* and *5*, the development flow calms down. In *Parts 4* there is still a bit of broad exploration (*i.e.,* the nodes in the graph are far away, symptom of a pure exploration phase). In *Parts 5*, instead, the working set is very narrow, nodes are mostly dark and very close between themselves. This means that the cognitive jumps are all focused on the current working set.

In the sessions exhibiting the “The Working Funnel” pattern, the working set is large at the beginning and progressively narrows down towards the end of the session, to guide the development flow, like a funnel.[3]

[3] A *funnel* is a pipe that is wide at the top and narrow at the bottom, used for guiding liquid or powder into a small opening.

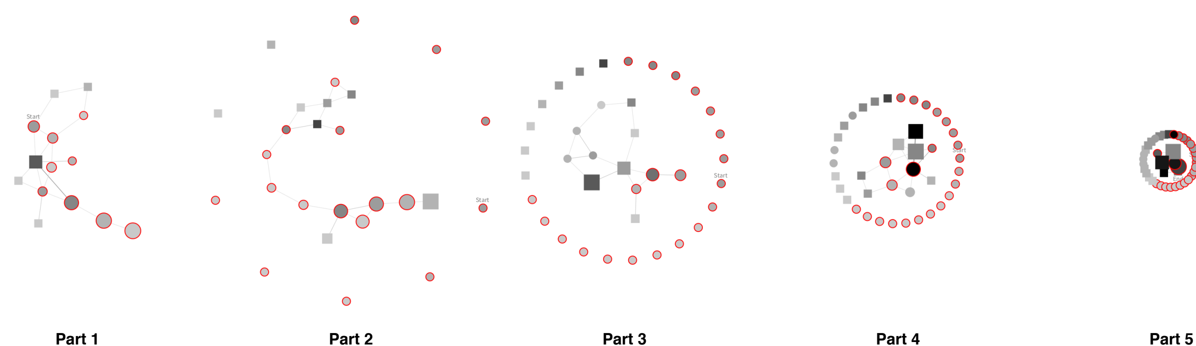


Figure 10.12. Another example of the “The Working Funnel” pattern

10.3 Reflections

In this chapter we presented a visual approach to characterize how working sets evolve during a development session. Our approach leverages navigation, edit, and inspection events, and provides an incremental, evolutionary visualization of the current and past working set. The view combines two different and dedicated layouts: A radial layout for the past working set, and a force-directed layout for the current working set.

We believe that visualizing interaction histories is an essential means to provide a deeper understanding of this novel and complex source of information. By inspecting our visualization, we identified several static and evolutionary patterns. In particular, we illustrated how our visualization can identify long, repeated navigations involving several entities, or the presence of key entities in a development sessions (that we called “guiding stars”) that guide development tasks. On the evolutionary side, we identified patterns where entities on the past working set return to be subject of tasks later in the session (*i.e.,* “The Past Awakens”), or cases where sessions are really composed of several independent tasks (*i.e.,* “Multi-Part Session”).

While reflecting on the characteristics of working sets is interesting to get insights about the mechanics of software development, we believe that our visualization can be more beneficial if integrated in the IDE and made actionable. As part of our future work, we envision developers that use our visualization as an alternative—and potentially faster—way to navigate among program entities.

11

Other Visualizations and Storytelling

The main motivation for our research is the fact that we believe that interaction data recorded by DFlow may reveal important insights about developer behavior inside the IDE. In the long term, this insights can be used to support the workflow of developers. However, after we developed the first prototype of DFlow and shared it with the first handful of developers, we discovered that raw interaction data are very hard to interpret.

To get a preliminary understanding of this data we devised a catalog of software visualizations that portrays interaction histories from different perspective. Our catalog includes visualization showing structural information of the code developers interact with or cumulative views to understand the impact of different activities (*e.g.,* navigation, inspection, editing) on a development session. This chapter details the catalog and explains how these visualizations can support visual storytelling of development sessions. In particular, we report two development stories that use multiple visualizations to gather interesting insights from the recorded development sessions. We applied the catalog visualizations on a dataset of more than 200 development sessions totaling 100,000 development activities (*i.e.,* meta-events) and about 80,000 interactions with the UI of the IDE (*i.e.,* window events).

This chapter also overviews DFloWeb, an early experiment we made on visualizing the workflow of developers in the web. The chapter details the web application, the visualization principles employed, and lists two peculiarities of the data emerged by a visual analyses of 20 development sessions recorded with DFlow.

Structure of the Chapter

Section 11.1 describes our catalog of visualizations to depict the developer behavior. Section 11.2 leverages such visualizations as the medium for visual development storytelling. In Section 11.3 we describe DFloWeb, our very first prototype of visualization of the workflow of developers. Finally, Section 11.4 concludes the chapter.

11.1 A Catalog of Visualizations for Development Sessions

Our catalogue of visualizations for development sessions includes 5 views: i) the Activity Forest, ii) the Activity Timeline, iii) the Cumulative Activity View, iv) the UI View, and v) the Workspace View. Since Chapter 9 already discussed the UI View, in this section we focus on the other views.

11.1.1 Activity Forest

The Activity Forest view depicts the program entities involved in a development session enriched with structural source code information. Figure 11.1 shows an example.

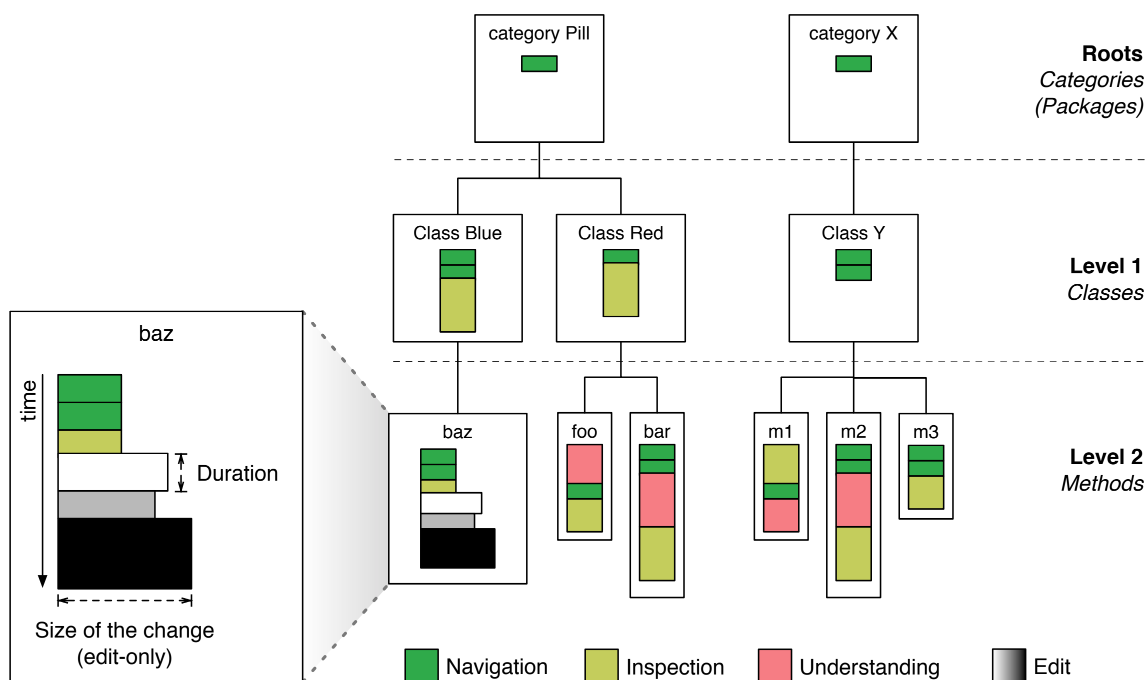


Figure 11.1. An example of the "Activity Forest" view

The visualization is composed of a forest of trees (two in the example), where each tree represents a sequence of development actions in a context, *i.e.*, subsequent actions happening on program entities contained in the same package. Thus, the root of each tree is a category (or package). Each category has classes as children. Only the classes subject to development actions are displayed (*i.e.*, not necessarily all the classes in the category). In the same way, classes have methods as children. Inside each node the view portrays development activities as colored boxes. The magnification on the left part of Figure 11.1 shows the activities on the method `baz`.

Each color represent a type of activity (see the color legend). The height of each activity box is proportional to the time spent, the width of boxes is fixed. The only exception are edit activities: Their width is proportional to the *size of the change*, *i.e.*, the difference between the size of the method before and after the edit. Edit activities are colored with a greyscale to represent the size of the method, *i.e.*, white for smaller methods and black for the biggest method edited in the session.

11.1.2 Activity Timeline

The Activity Timeline view portrays the development activities of a development session as a timeline. This view, depicted in Figure 11.2, emphasizes the sequential nature of the activities and their duration.

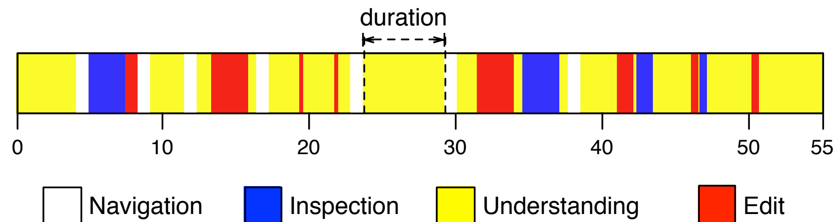


Figure 11.2. An example of the "Activity Timeline" view

In the example, each activity is represented by a colored box: white for navigation, blue for inspection, yellow for understanding, and red for editing. While the height of the timeline (*i.e.,* and of the activities) is fixed, the width of each activity is proportional to its duration. The timeline is enriched by regular time ticks at 10 minutes intervals.

11.1.3 Cumulative Activity

The Cumulative Activity View places development activities in a cumulative bar chart. This layout stresses the partitioning of different types of activities (*e.g.,* navigation, editing, understanding). Figure 11.3 shows an example.

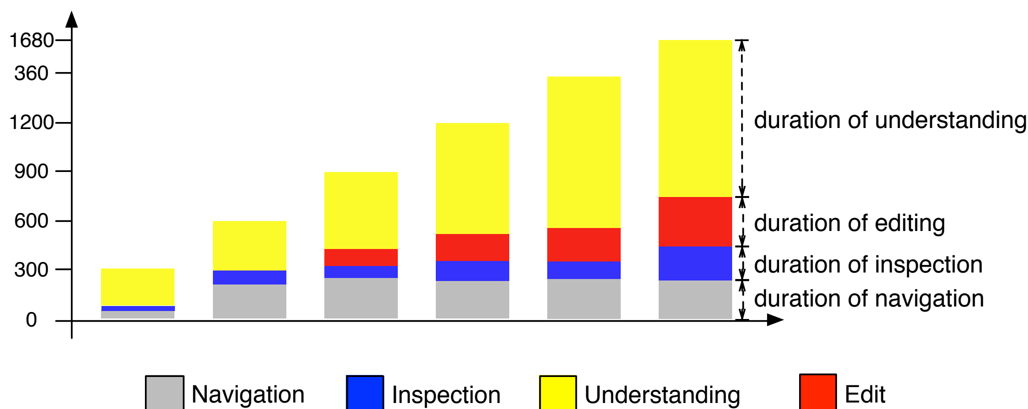


Figure 11.3. An example of the "Cumulative Activity" view

This visualization presents the same information of the Activity Timeline of Figure 11.2 but in a different form. The vertical axis of the graph represents time (in seconds). The first bar represents the activities in the first 5 minutes of the session, the second bar portrays the first 10 minutes, and so on.

11.1.4 Workspace View

The Workspace View mirrors the *Pharo* IDE and depicts position and size of opened windows over time. It highlights which areas of the IDE are the most crowded.

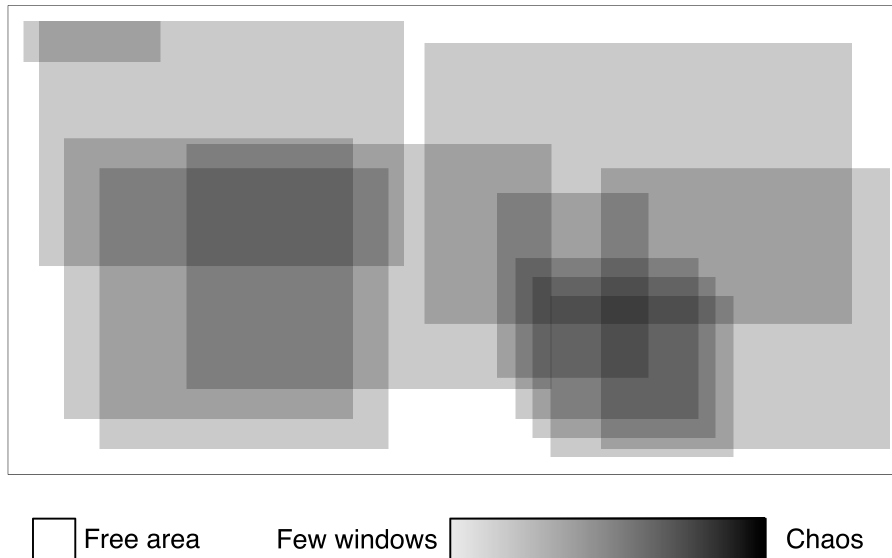


Figure 11.4. An example of the "Workspace View"

Figure 11.4 shows an example. The outermost container is the *Pharo* IDE. Inside there are translucent grey boxes representing the windows the developer interacted with during a development session. The view shows the evolution of the entire session, step by step. Figure 11.5 shows three subsequent snapshots of a session through the Workspace View.

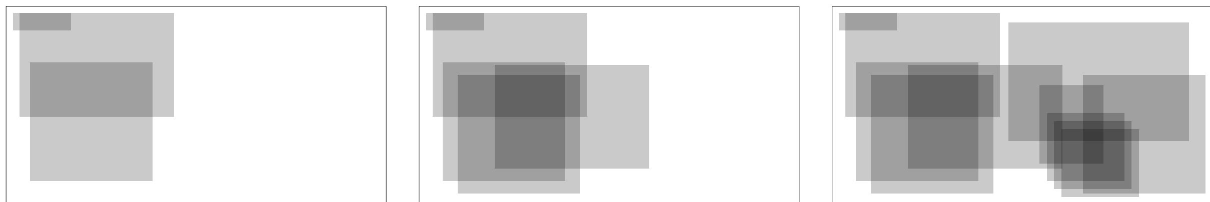


Figure 11.5. Subsequent moments visualized through the "Workspace View"

At the beginning of the session (see Figure 11.5) the developer concentrates her focus on the leftmost part of the IDE. As the session continues the developer fills the IDE with windows. The highest *"concentration of windows"* happens near the bottom right corner of the IDE (*i.e.,* darkest area of the view).

Summing Up

Our visualizations serve to characterize the behavior of developers during development sessions. The Activity Forest, Activity Timeline, and Cumulative Activity View depict development activities such as navigating, inspecting, editing, and understanding source code. The Activity Forest highlights the program entities involved in the development session and their source code structure. The Activity Timeline and the Cumulative Activity View, instead emphasize how time is spent while interacting with the IDE. The other two views, the UI View and the Workspace View, focus on pure UI interactions.

In the next section we put the visualization in practice to support visual storytelling of development sessions.

11.2 Telling Visual Development Stories

In this section we present two development stories emerged from the visual analyses of development sessions: i) “Killing Bugs and Windows”, and ii) “One Window Takes It All”.

Killing Bugs and Windows

The first story is about a session of a developer that we will call Alice. Upon starting a session DFLOW asks the user for a *title* and a *session type*. The developer categorized the session as *bug-fixing*. The session lasts for about three hours, including 1 hour of pause. Figure 11.6 shows how the developer managed her time in terms of development activities.

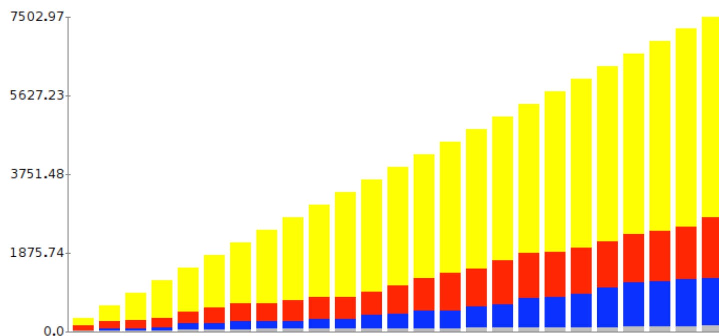


Figure 11.6. Cumulative Activity View for a bug-fixing session of Alice

In each of the 5 minutes slices the developer mainly performed understanding activities. At the end of the session understanding accounts for 1 hour and 20 minutes, editing activities lasted for less than 25 minutes, and duration of inspections and navigations are respectively 19 and 2 minutes. The large predominance of understanding could be intrinsically connected with the nature of the session: Bug-fixing requires a deep knowledge of the code base.

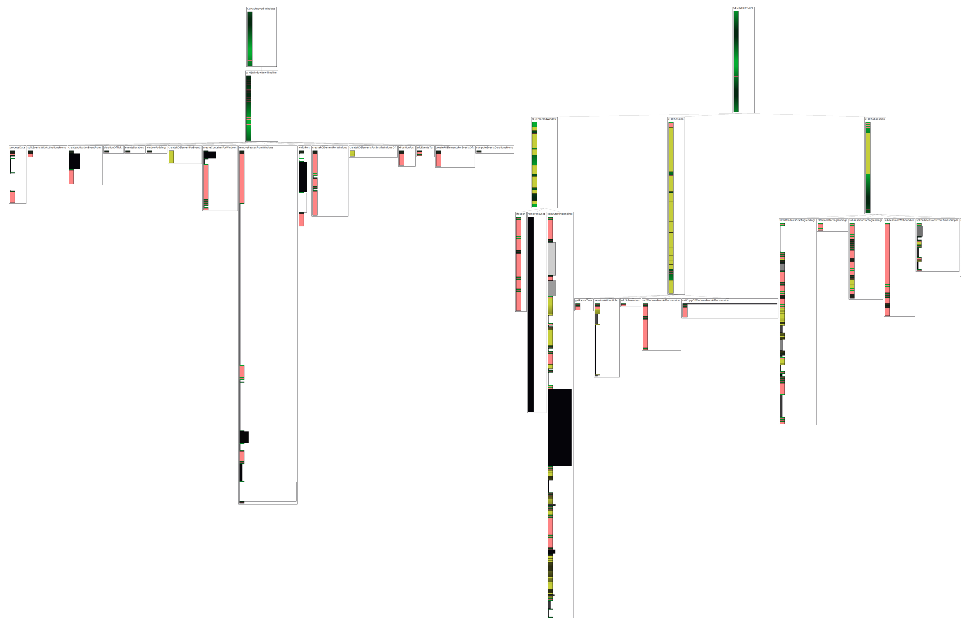


Figure 11.7. Part of the Activity Forest for a bug-fixing session of Alice

The half hour the developer spent in editing source code can be summarized with the part of the Activity Forest (see Figure 11.7). All the edits are condensed in two contexts (*i.e.,* packages) and involve only a dozen methods. Most of the times edit events are interleaved with inspections (depicted in pale yellow), which are the means to understand the effects of the changes.

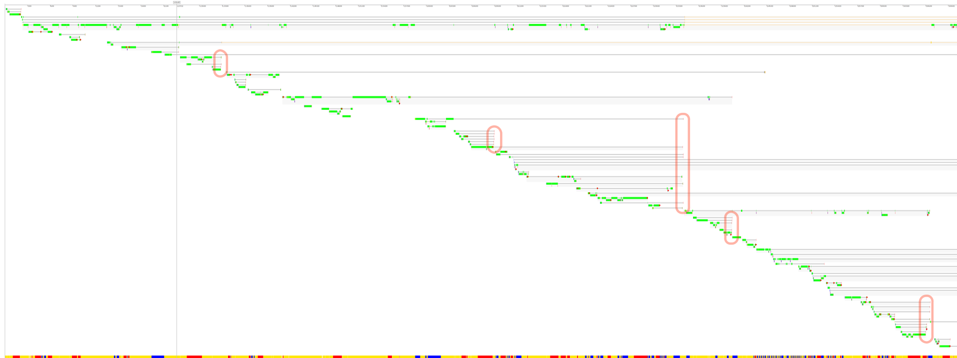


Figure 11.8. Combined UI View and Activity Timeline for a bug-fixing session of Alice

Until now we only focused on development activities. DFLOW also captures interactions with the UI of the IDE. Figure 11.8 shows a combined visualization of the UI View (top) and the Activity Timeline (bottom) for the same Bug-Fixing Session of Alice. In this session Alice used 228 windows and she focused for very little time on each of them (*i.e.,* about 30 seconds per window). The highly interrupted development's flow of Alice in this session may be due to the way the IDE supports debugging activities. In Smalltalk, while debugging, developers perform inspections on instances of objects. Most of the times when the user inspects an object the Pharo IDE triggers an *Inspector*, *i.e.,* a small window that shows details about the inspected instance. This assumption is supported by a high number of inspection events (222). This session features 60 edit events on a dozen of methods. The red dots in the UI view represent when and where edit events happened. From Figure 11.8 we can observe that there are more than a dozen windows with edit events. Alice tends to open multiple source code browsers on the same artifact, and close browsers immediately after an edit, thus being forced to reopen it for the next edit.

Developers are often forced by IDEs to spawn a number of windows (or tabs) to reveal hidden relationships among source code entities. Röthlisberger *et al.* called this phenomenon the *window plague* [RND09]. From the highlights in Figure 11.8 we can observe how the environment of Alice is affected by this plague. When her IDE reaches a certain "level of chaos" she simultaneously closes a number of window to lower it. Figure 11.9 shows two snapshots of the session of Alice using the Workspace Views. Alice has a tendency to use only the leftmost part of the IDE. This could possibly motivate the need to cure the *window plague* so often.

Figure 11.9. Two Workspace Views of the bug-fixing session of Alice

One Window Takes It All

This story is about an *enhancement session* of Bob. Enhancement means that the developer's intention is to add new or enhance existing functionality.

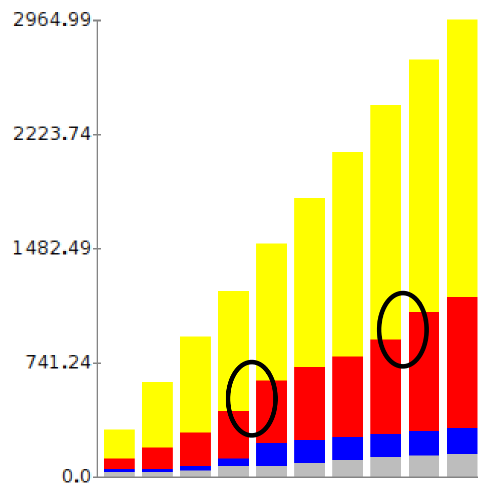


Figure 11.10. Cumulative Activity View of the enhancement session of Bob

The session lasts for about an hour and for its entire duration the developer spends the majority of her time in understanding (see Figure 11.10). Editing increases constantly throughout the session with two major jumps, highlighted in the view.

All the edits happen in a single category (or package) as shown by the part of Activity Forest depicted in Figure 11.11. It highlights the activities on two methods, both part of the same class, *i.e.,* the most edited methods. From the visualization we see that the developer tends to shorten these methods while editing them, *i.e.,* their color goes from black to white. The complete Activity Forest (not shown for lack of space) includes a number of small trees depicting classes the developer browsed while building his knowledge to perform the changes.

Figure 11.12 shows how the IDE looks like at the beginning, towards the middle, and at the end of the session. There is a big window (*i.e.,* a code browser) that occupies almost the entire IDE space. This window remains active for the entire duration of the session. The developer tends to open (or move) all the windows towards the top let corner of the screen, hiding the top half of the big window. Pharo code browsers display source code in the lower part of the window. Bob moves all the windows so that he can always see the lower part of the big-window, most likely because he wants to keep an eye on the source code displayed in it. The UI View, depicted in Figure 11.13, shows this long-lived window, *i.e.,* the first window track. All the edits are performed using this window, *i.e.,* this session revolves around this key window.

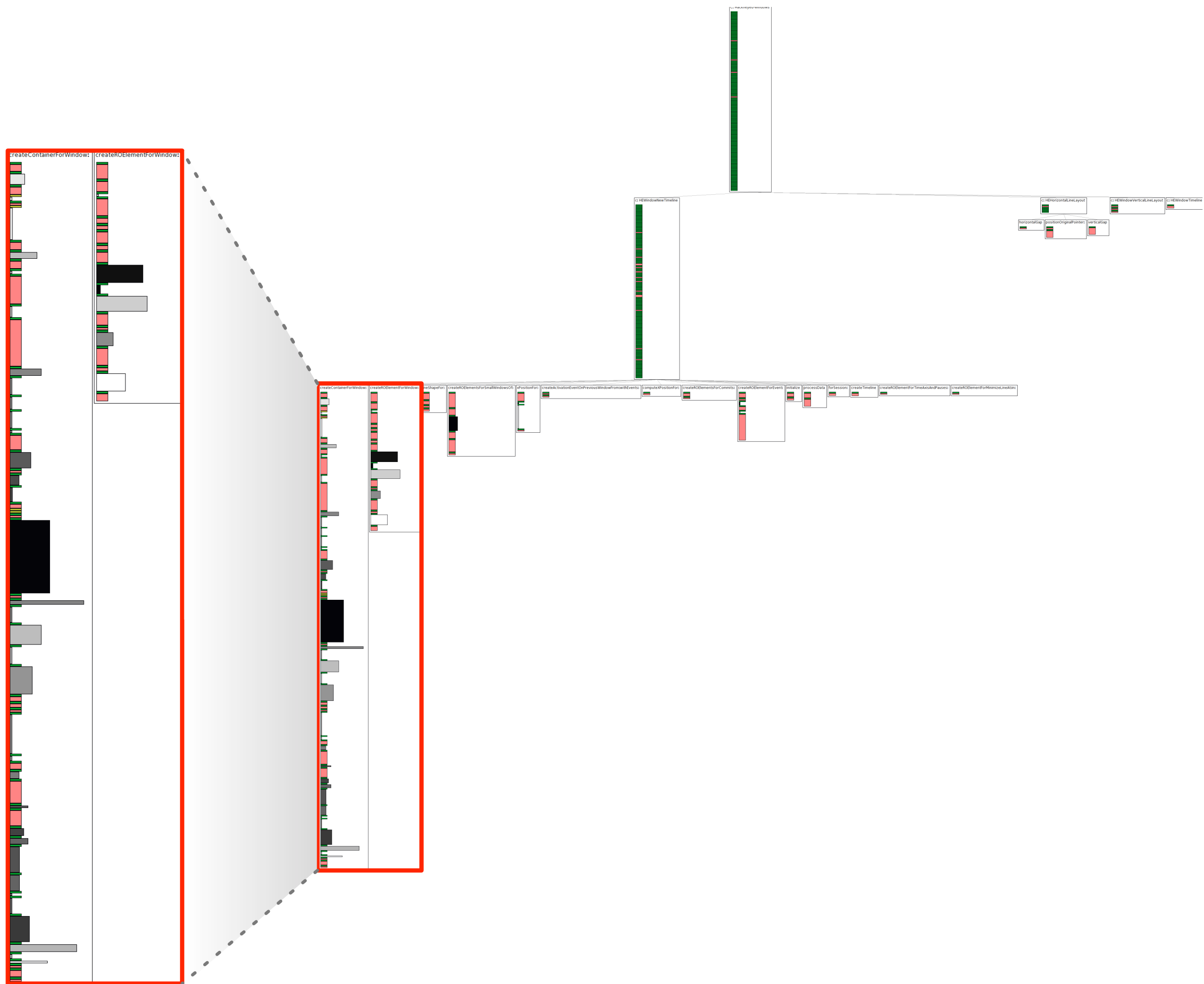


Figure 11.11. Part of the Activity Forest of the enhancement session of Bob

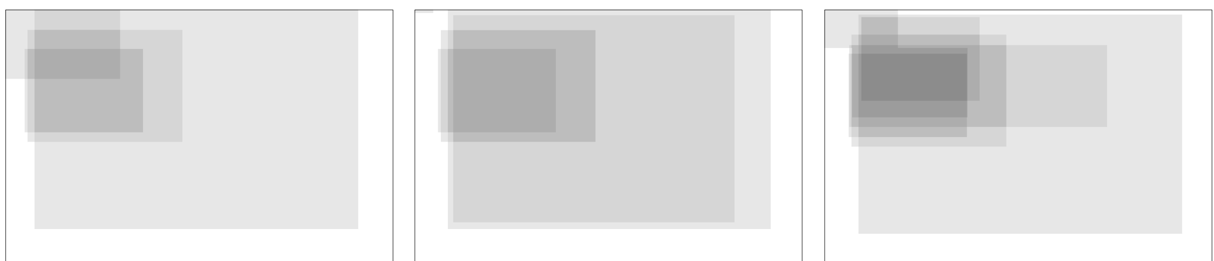


Figure 11.12. Three Workspace Views of the enhancement session of Bob

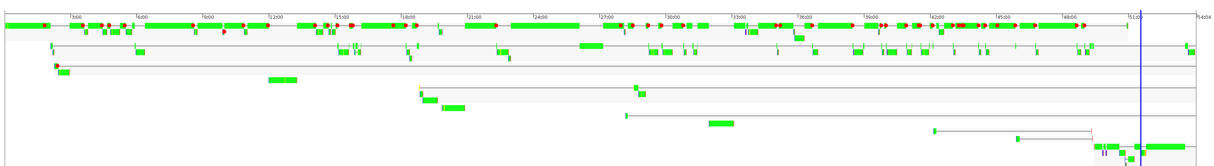


Figure 11.13. The UI View of the enhancement session of Bob

11.3 DFloWeb: Visualizing Interaction Data in the Web

DFLOWEB, depicted in Figure 11.14, was one of our early attempts at visualizing interaction data. This visualization platform is implemented as a web application.

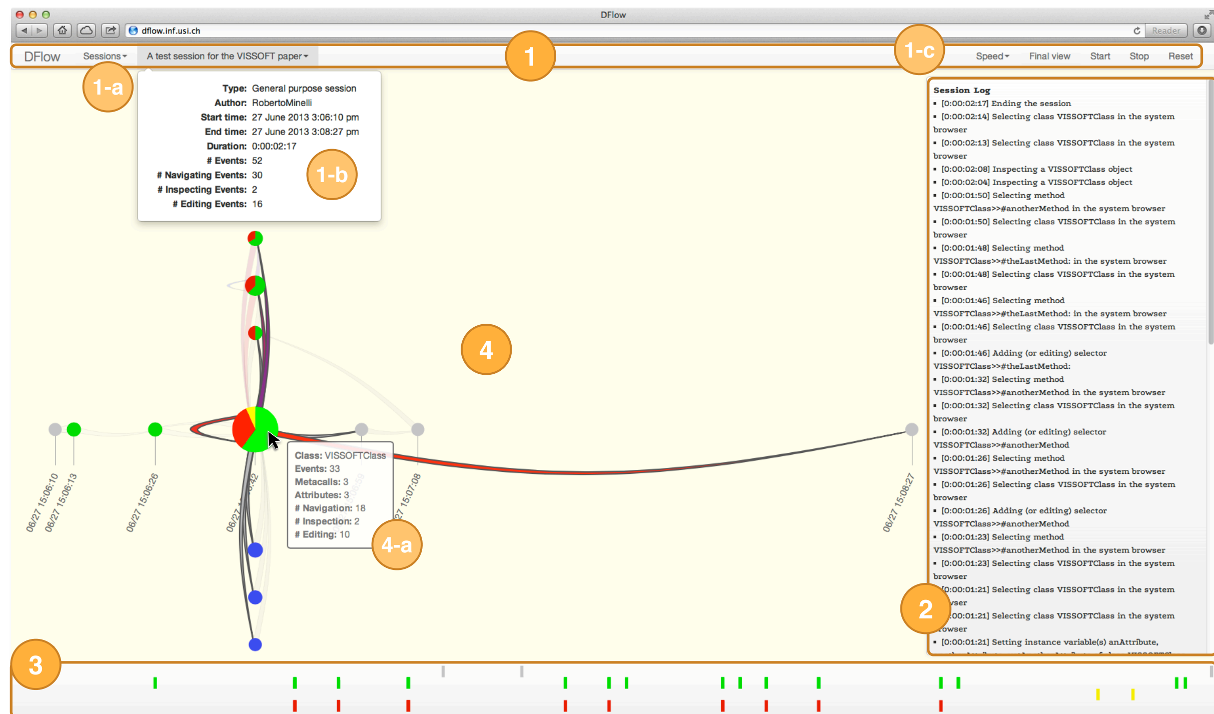


Figure 11.14. DFloWeb composed of (1) a Navigation Bar, (2) a Session Log, (3) a Timeline, and (4) the Visualization Canvas

The web application of DFLOWEB is composed of:

1. **Navigation Bar** to configure the visualization and browse information.
   (a) *Select Session Menu* to select the session she wants to analyze.
   (b) *Session Information Panel* to provide additional information on the session.
   (c) *Replay Menu* to step into the session (*i.e.*, start & stop) and to configure the speed of the animations.

2. **Session Log** to show a time-ordered textual description of each event that happened during a session.

3. **Timeline** to represent the events happening in a session divided according to their category: handling, navigating, inspecting, and editing.

4. **Visualization Canvas** for the interactive visualizations.
   (a) *Entity Information Panel* to reveal additional information about the hovered entity.

11.3.1 Visualizing Development Sessions with DFloWeb

DFloWeb uses a custom visualization to depict a development session. Figure 11.15 shows the principles behind the view and the color mappings we use for the entities. We depict a development session using a *directed graph* in which nodes are the entities involved in the current session (classes, methods, attributes). The directed links (*i.e.,* source destination) depict "navigation-paths" and not structural relations.

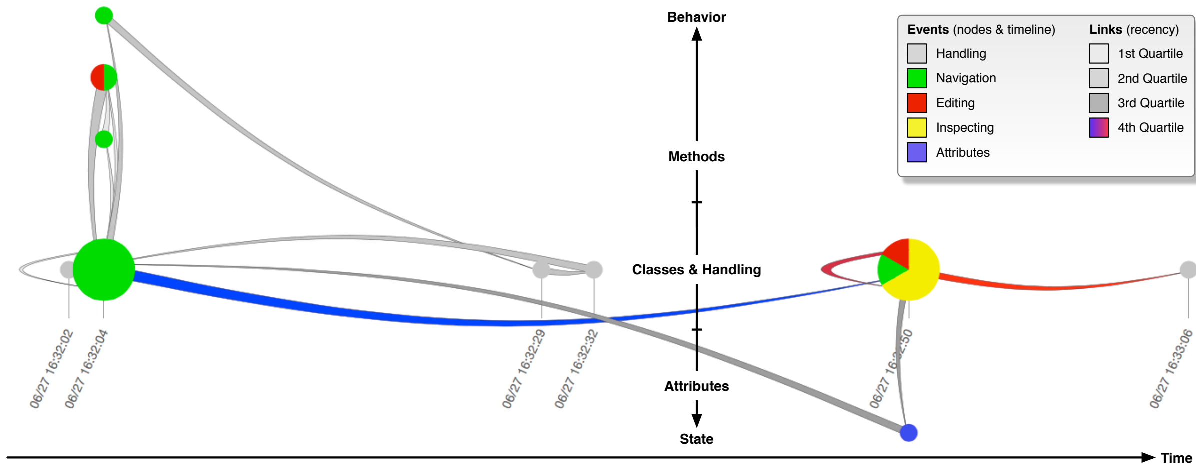


Figure 11.15. Visualization principles of DFloWeb

Principles and proportions

Nodes are either classes, methods, attributes, or session handling events (*e.g.,* start, pause, stop, resume). The radius of a node is proportional to the number of events on that entity in the sessions (*i.e.,* how many times the user interacted, directly or indirectly, with this entity). We divided events in three categories: navigating (*i.e.,* green), inspecting (*i.e.,* yellow), and editing (*i.e.,* red). Navigation events are the less intrusive events (they do not modify the entity) while editing events represent the "real" editing activities (*e.g.,* adding/modifying a method/class). Inspection events do not modify the entity, but represent deeper forms of navigation (*e.g.,* inspecting the internals of an object). Class and method nodes are depicted as pie charts presenting the event distribution of that entity at a glance. Handling events are depicted in grey. Links depict "navigation paths" between entities, *e.g.,* if the developers creates `Class A` and right after browses the Method `foo` of Class `B` (*i.e.,* `B#»foo`) DFloWeb draws a directed link from `Class A` to `B#»foo`.

The width of the link is proportional to the number of occurrences of that navigation path in the session. We use the color to present information about the age of the links. We divided links in quartiles, according to their age. For the first three quartiles (*i.e.,* old links) we used three tones of gray with increasing saturation, while for the last quartile (*i.e.,* the most recent links) we use a gradient from blue to red.

Class nodes and session handling nodes are positioned on a horizontal line, whose x-coordinate represents the temporal dimension of a session. Method nodes and attributes are not mapped to that time scale, but they take the x-position of their owner node (*i.e.,* a class node), unless the owner node is not part of the view. This visual cue helps one to perceive to what extent a class has been touched during a session.

The bottom timeline presents an outline of the session, where each rectangle is an event. The color of events follows the same color scale of the pie-charts. The y-coordinate of each rectangle represents one of the four categories of events (*i.e.,* handling, navigation, inspection, and editing). The x-coordinate of each rectangle represents the timestamp of the event it depicts.

Interacting with DFloWeb

The user can interact with DFLOWEB by panning (*i.e.,* drag & drop) and zooming (*i.e.,* mouse wheel) the view inside the visualization canvas (Figure 11.14.4). The zoom performs an x-axis rescale and restricts the time interval being displayed. This helps to better understand the visualization at time steps where events have a high density. The user can also drag & drop single nodes to better understand links between the nodes. The "Replay menu" (Figure 11.14.1-c) offers additional means to interact with the view. DFLOW records all the time steps of a session, and DFLOWEB is able to produce an animation of the session where the view evolves together with the session. Figure 11.16 shows three time steps from the evolution of a session.

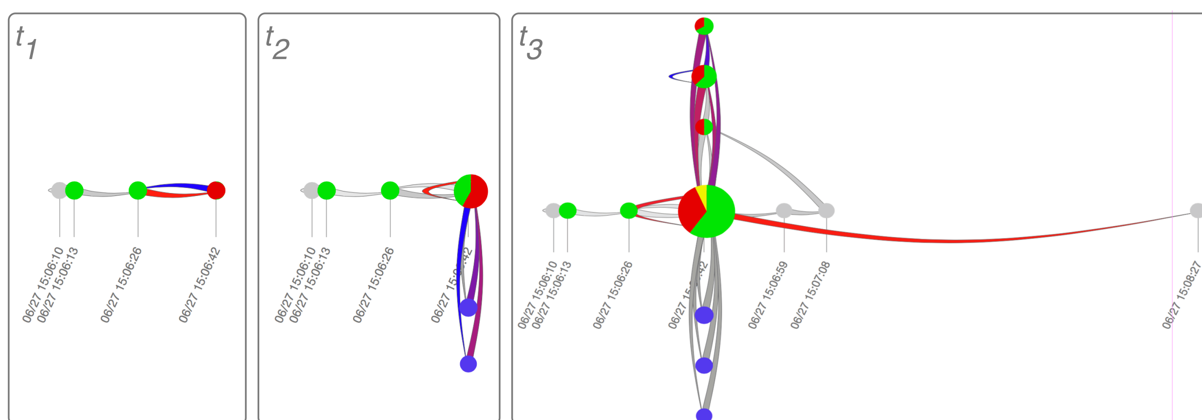


Figure 11.16. Three time steps of the same session visualized with DFloWeb

11.3.2 Telling Development Stories with DFloWeb

We analyzed 20 sessions with DFLOWEB. The analysis uncovered two main insights:[1]

- "High Navigation Stacks & Back-Links", and
- "Different Type, Different Shape".

[1] To increase the readability of this Section, we present each insight on a new page.

High Navigation Stacks and Back-Links

Figure 11.17 shows part of an "enhancement session" recorded during the development of DFLOW itself. There are 3 main stacks of events highlighted in the Figure. Stacks A and B refer to user-defined classes, `DFSessionAnalyzer` and `DFJSONTouched-EntityNode`. Stack C is a chain of navigation events involving the `String` class and its methods. High navigation stacks denote that the developer is browsing the API of some class to find a specific piece of functionality.

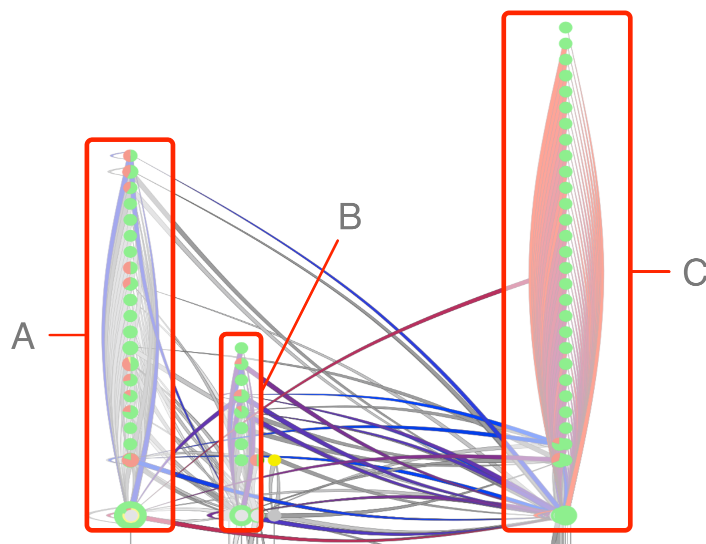


Figure 11.17. A Fraction of an Enhancement Session Depicted with DFloWeb

Figure 11.18 shows a snapshot of the same session at a later time. In the figure we marked two special links, A and B. These are "back-links", where the developer, after repeatedly browsing some other class, has gathered enough knowledge to "go back" to another entity and finally perform an informed modification.

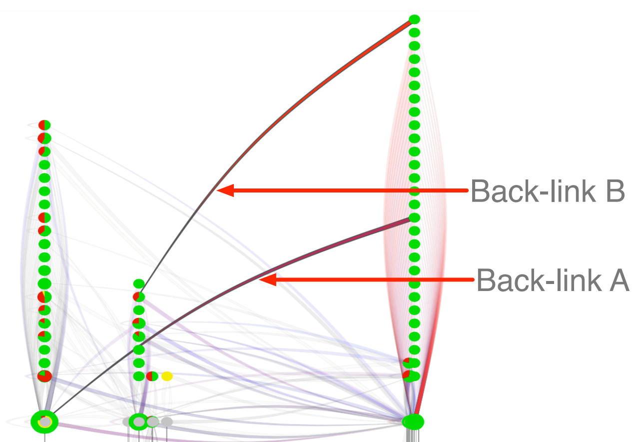


Figure 11.18. The Same Session of Figure 11.17 at a Later Time

Different Type, Different Shape

Figure 11.19 shows a "bug-fixing session". From both the bottom timeline, and the visualization, we see that this session features a series of navigation events (*i.e.,* green) and one single editing (*i.e.,* red). This denotes the fact that the user has been navigating code to gather information about the system to fix one specific entity, marked as A.

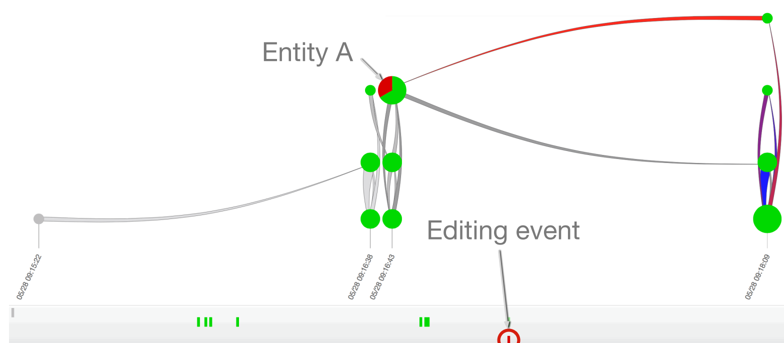


Figure 11.19. DFloWeb Depicting a Bug-Fixing Session

11.4 Reflections

Often times it is unclear how developers exploit the diverse facilities offered by IDEs to perform their activities. We believe that software visualizations are an intuitive mean to gather insights from data that are otherwise hard to interpret. In this chapter we presented a catalogue of visualizations to support basic analytics of developer interactions with the IDE. We tested our visualizations to support visual storytelling of interesting developer behaviors. The two stories illustrate that it is possible to infer insights about how developers use the IDE, pointing out veritable development styles in terms of UI usage. The chapter also detailed DFLOWEB, a web application that we developed in the early stages of our research to visualize the workflow of developers. DFLOWEB enables retrospective analyses through interactive web-based views.

All the visualizations described in the last chapters share one limitation: They can only be used retrospectively. We envision "*live and adaptive visualizations*", visualizations that are in-sync with the workflow of developers (*i.e.,* live) and that can be used by developers as a support for the development process. In Chapter 14 we further discuss this long term vision.

Part IV

Supporting Developers with Interaction Data

12

The Plague Doctor: Curing the Window Plague

Object oriented programming introduced a number of benefits in terms of how software systems are developed, structured, and organized: Better separation of concerns, modularity, and reusability are mere examples. However, this comes at a cost: Program entities are organized in hierarchies, stored over complex repositories, and thus there can be complex, hidden, and transitive relationships among them [WH92, DRW00]. This hampers program comprehension that, among other objectives, aims to explore and understand such complex relationships.

IDEs provide two main UI paradigms: *window-based*, like in the *Pharo* IDE, or *tab-based*, like in the *Eclipse* IDE. Neither of the two paradigms effectively supports the navigation of the complex software space [SES05]. In fact, both paradigms force developers to open one tab (or window) per program entity, leading to what researchers called *window plague* [RND09]. The *window plague* is the tendency of IDEs to quickly become overcrowded by unused windows (or tabs). In addition, IDEs do not keep track of relationships among windows and provide little or no support to automatically maintain a low level of entropy inside the IDE, *e.g.,* by closing unused windows.

It has been shown that it is possible to mitigate the *window plague* by monitoring how developers interact with the UI of the IDE and exploiting such data [RND09]. Röthlisberger *et al.* developed a preliminary cure for the *window plague* in Autumn Leaves. Autumn Leaves is an extension for the *Pharo* IDE that detects windows that are unlikely needed for further use and closes them. It also adds visual clues to the more important ones to provide cognitive feedback to the IDE. The authors conducted a benchmark evaluation with promising results. Unfortunately, Autumn Leaves remained a prototype, it was never integrated in the IDE and no one could take advantage of its potential benefits. We believe that one of the reasons for this was overly coarse grained data leveraged by Autumn Leaves, and it remains an open issue to quantify how much the *window plague* hinders development.

The *window plague* is a relevant problem for the *Pharo* community, in this chapter we present the Plague Doctor: an enhanced implementation of Autumn Leaves. In this chapter we also discuss possible future directions to provide a more effective cure for the *window plague*.

Structure of the Chapter

Section 12.1 details the approach and its implementation. Section 12.2 outlines possible future directions for research in this context and Section 12.3 concludes the chapter.

12.1 The Plague Doctor

The top left part of Figure 12.1 shows the *Pharo* IDE manifesting the *window plague* after a few minutes of development. The lower right part of Figure 12.1 depicts the same environment with the PLAGUE DOCTOR enabled. Without the PLAGUE DOCTOR, the UI depicted in Figure 12.1 is overcrowded by a significant number of apparently similar, overlapping windows and makes impossible for developers to identify which windows are most relevant for the current development context. It is also likely that some of these windows are not needed, but the IDE does not provide means to automatically identify them or reduce the level of entropy, *i.e.,* by closing windows.

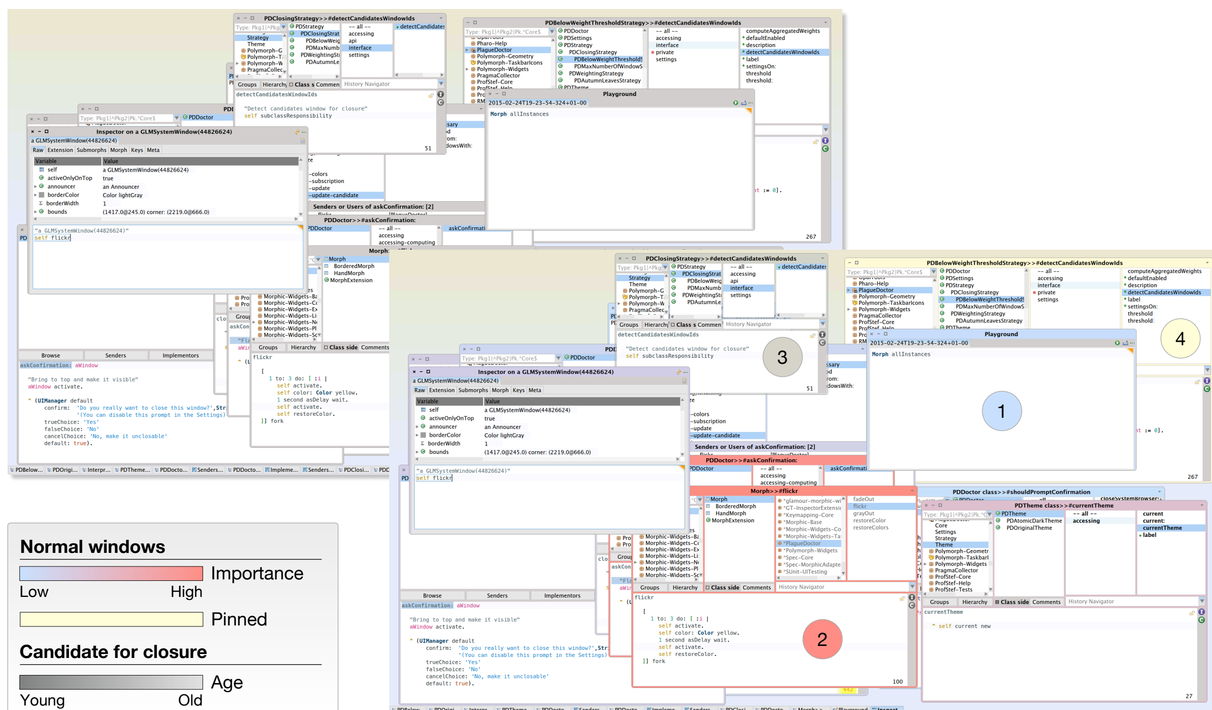


Figure 12.1. A screenshot of *Pharo* IDE manifesting the Window Plague (top left) and the same environment after enabling the Plague Doctor (bottom right)

The PLAGUE DOCTOR uses interaction data to compute the importance of windows, and thus the likelihood that they will be used again in the future. For example, a window becomes more important when a user types on it, or when she uses its UI components to perform a task. It decorates the windows to keep the developer aware of their computed importance. A heat-scale from light blue (*i.e.,* less important, see Figure 12.1.1) to bright red (*i.e.,* more important, see Figure 12.1.2) reduces the cognitive load of a developer that faces this IDE: It is likely that she can concentrate her focus on the most warmer (important) windows and ignore the colder windows. In addition to the color scale for the important windows, the PLAGUE DOCTOR lets the user identify the windows that are less likely to be used in the future. When less important windows become candidates for closure, the PLAGUE DOCTOR uses a gray scale from dark to bright gray to indicate their age, *i.e.,* in terms of how many interactions happened after the window became a candidate (see Figure 12.1.3). By default, the PLAGUE DOCTOR has a grace period of 5 user actions before closing a window that has been marked as a candidate. When this period expires, the doctor automatically closes the window, asking for user confirmation if configured to do so.

For some tasks, the developer may become aware that an unused window is still important for a future task, and thus she might require to avoid its closure in an explicit manner. The Plague Doctor provides the ability to *pin* a window, that is, exclude it from the potential candidates for closure. Pinned windows are colored in light yellow (see Figure 12.1.4).

The Plague Doctor is just the tip of the iceberg. While the developer programs, DFlow observes all the user interactions, from UI events such as moving a window, to meta events such as the creation of a new class, down to the granularity of mouse and keyboard events. DFlow then generates events that other tools, such as the Plague Doctor, can intercept, process, and exploit, as discussed in Chapter 5.

12.1.1 Models and Strategies

The Plague Doctor defines the *"importance"* (or weight) of a window in the current development context. To compute it, it maintains two weight models: the *window interaction model* and the *program entity model*. The global weight of a window, is computed by combining its weight from the window interaction model and the weight of the program entity displayed in the window itself (if any). To update these models the doctor uses a weighting strategy. Closing strategies, instead, determine which windows are candidates for closure. The user selects one weighting strategy and one or more closing strategies, *i.e.,* we call them *active* strategies. After every interaction the active strategies are applied: Models are updated, windows are decorated and closed, if needed.

Program Entity Model

The Program Entity Model associates a weight to each program entity (*i.e.,* class or method) observed during a development session. Every time the developer interacts with an entity (*e.g.,* observe, modify) its weight gets updated, according to the defined weighting strategy. The weight of a program entity is persisted even if all the windows that display that entity get closed. This allows the doctor to keep track of the entities that are relevant in the current development session.

Window Interaction Model

Similarly, the Window Interaction Model associates a weight to each open window during a session. The weight gets updated at each interaction with the specific window (*e.g.,* on window focus, minimization, movement), and depends on the active weighting strategy. When a window is closed, its weight is removed from the model.

Weighting Strategy

A Weighting Strategy determines how weights are updated. In the original Autumn Leaves strategy, every user interaction brings a particular, fixed, weight update. The doctor implements this strategy, and uses the original parameters and weight updates suggested by Röthlisberger *et al.* [RND09]. However, we will investigate the effectiveness of the original parameters. The original strategy prescribes that 50% of the weight updates of program entities is propagated following structural source code relationships (*i.e.,* method propagates to its defining class, class to its direct superclasses and subclasses). Currently, only one weighting strategy at the time could be active.

Closing Strategy

A Closing Strategy is responsible to determine which windows are candidate for closure. For example, a strategy that involves the weight models can define a threshold (*i.e.*, sensitivity) on the weight of windows. As in the case of AUTUMN LEAVES, we implemented a strategy that closes all the windows whose weight is below a customizable threshold. The default approach uses a percentage of the average weight of all the windows. An innovation with respect to AUTUMN LEAVES is that the user can activate more than one closing strategy at the time. A strategy could consider the weight models or ignore them, *e.g.*, one might want to have a maximum number of open windows per window type. In an IDE, often, there are different *kinds* of windows: workspaces, code browsers, test runners, *etc.* We also implemented a strategy that closes the windows with lowest weight of a given type, when the IDE reaches the maximum amount of open windows for that type.

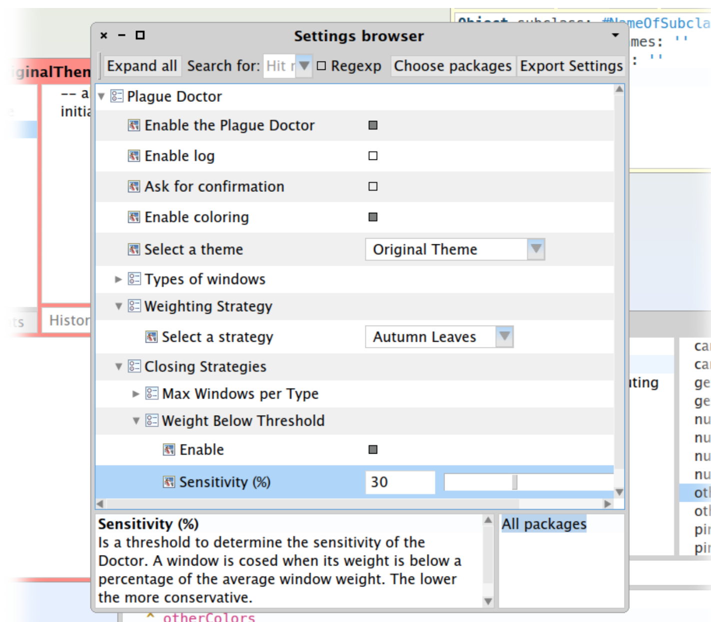


Figure 12.2. The Settings of the Plague Doctor

12.1.2 Advocatus Diaboli

We are aware that the PLAGUE DOCTOR has some limitation. This section addresses some criticisms that can be raised against our approach.

It only works for window-based IDEs. Even though our prototype has been implemented in *Pharo*, a window-based IDE, the approach is not specific to such environments. The default weighting and closing strategies can be applied directly to tab-based IDEs such as *Eclipse*.

The Plague Doctor does not differ from Autumn Leaves. Our tool is indeed inspired by AUTUMN LEAVES, and mimics all its functionalities, but it has a number of advantages:

- It exploits **more** and **better** interaction data. The doctor can leverage all the fine-grained data recorded by DFLOW. For example, using the code browser to navigate source code

might increase the weight of a window and all the visited entities. Debugging events can be leveraged to increase the weight of the entities being investigated.

- It is **extensible**. The Plague Doctor is designed to be extensible. For example, adding a new weighting (or closing) strategy requires minimal effort, *i.e.,* a new class copied from a template and a method that implements the strategy per se. The new strategy will immediately appear in the settings of the Plague Doctor (depicted in Figure 12.2) and can replace the current one right away.
- It is **customizable**. The Plague Doctor has a number of settings, depicted in Figure 12.2, that the developer can use to customize it. For example, all the colors are contained in a theme class that can be duplicated and changed to have a novel and more appealing color scheme.
- It is **Available**. Differently from Autumn Leaves, the Plague Doctor is currently available and can be installed in the *Pharo* IDE.

12.2 The Future of the Plague Doctor

The prototype of the Plague Doctor described in the previous section is only the first step towards fully exploiting the data collected by DFlow while the developer is programming. This section discusses our future plans to provide a more effective cure for the *window plague* leveraging DFlow data.

Fine-tuning the existing strategies

Weighting and closing strategies are parametrizable. The value of the weight update after a particular user interaction, for example, is a parameter of the weighting strategy. The usefulness of the Plague Doctor strictly depends on how good the strategies are. Until now, we reused the values proposed by the authors of Autumn Leaves [RND09]. The authors used a benchmark evaluation to devise such values. We are in contact with the core developers of the *Pharo* community, and we plan to conduct a detailed user evaluation with them to fine-tune these parameters backing them up with evidence from interaction data.

Adding Novel Strategies

Our initial Plague Doctor prototype makes it easy to add new strategies to weight or close windows in different ways. Our plan is to devise a number of different strategies and test them in real settings scenarios, *i.e.,* involving real developers. A qualitative study where we can get feedback from developers about our approach could also trigger new ideas for novel strategies. Since the Plague Doctor allows multiple closing strategies to be enabled, we should also investigate which combinations of strategies perform better.

Self-Adaptation

Our long term vision focuses on the *self-adaptability* of IDEs [Min14]. We believe that tools should also be subject to self-adaptation. In this context, for example, strategies could be self-adaptable. Consider the closing strategy that uses a threshold to decide which windows to close. Suppose that, when the developer realizes that the Plague Doctor wants to close a window, she *pins* that window to force the doctor to leave it open. The doctor must lose confidence in

itself and relax its sensitivity. In the opposite case, if the doctor closes windows without the user playing against, it should slightly increase its confidence and increase its sensitivity.

Exploiting more interactions

The Plague Doctor currently exploits only a few more user interactions with respect to Autumn Leaves. Potentially, it can leverage all the fine-grained interactions collected by DFlow [MMLK14]. Mouse events, for example, can be factored in the weighting strategies. In fact, developers might use the mouse as a *reading device*, *i.e.,* by following the source code that they are reading with the mouse cursor. Another source of information collected by DFlow are debugging events. If a developer spends time in debugging a piece of code, it is likely that the program entities contained in this code snippet are relevant for the current development session, thus their weights should increase.

Evaluation Plan

To validate the Plague Doctor, we plan to do a benchmark evaluation and feed the recorded sessions to the tool. The idea is to define the level of entropy of the IDE (*i.e.,* how many unused windows are left open) and measure if and how it varies with the support of the Plague Doctor. We also aim to obtain further evidence of the importance of the *window plague* in practice. Our expectation is that the tool is effective to reduce the level of entropy while being as precise as possible. By precise we mean that the Plague Doctor should only close windows that the developer would not reuse in the future. There is a trade-off between precision and effectiveness that remains to be investigated and optimized. Another study could focus on the time spent by developers in program understanding tasks. From a previous analysis of recorded interaction data, we found that developers spent a considerable amount of time (ca. 15%) in fiddling with the UI of the IDE (*e.g.,* by rearranging windows that create confusion in the IDE). Since the *window plague* is one possible reason behind this, we should investigate if approaches such as Autumn Leaves or the Plague Doctor reduce the time wasted by developers in fiddling with the UI of the IDE. Last but not least, we also plan to conduct a qualitative evaluation to gather direct feedback from developers.

12.3 Summing Up

IDEs offer little support to navigate the complex and implicit relationships among program entities and force developers to open one window (or tab) per entity, leading to what researchers called *window plague*, an overly crowded workspaces with many open windows (or tabs) [RND09].

In this chapter we presented the Plague Doctor, a tool that leverages fine-grained interaction data collected by DFlow to mitigate the *window plague*. Our tool computes the importance of windows, decorates them to reduce the cognitive load of a developer facing the IDE, and closes the windows that are less likely to be used again in the future. Plague Doctor is only a prototype. In this chapter we discussed its limitations and our future plans towards a more effective cure for the *window plague*.

13

Taming the User Interface of the IDE

The window plague, discussed in the previous chapter, is one of the reasons that can lead the IDE in a "chaotic"[1] state [RND09]. In general, modern IDEs are not able to effectively and efficiently tackle the complex relationships between source code entities to support source code navigation. Researchers discovered that up to 35% of development time is spent navigating code [KMCA06] and that developers often need to periodically revisit the same entities [SKG+13]. Since there is limited evidence that a chaotic IDE state affects the developer's productivity ([BRZ+10]), the nature and amount of chaos experienced by developers is hard to characterize and quantify. This makes developing mitigating countermeasures challenging.

In this chapter we look for evidence of chaos by analyzing two different datasets. The first one comes from *Pharo*, a window-based IDE, and includes around 770 hours of development data coming from 17 developers. We observe that these developers spend on average 30% of their time in a chaotic environment. The key characteristic that makes our approach possible is the large number of fine-grained interaction data that we were able to record over an extended period. To get a more encompassing vision, we also analyzed the Mylyn dataset [KM05], which features several thousands of individual tasks coming from 179 developers.

With the fine-grained dataset of DFlow, we were able to comprehensively investigate the chaos phenomenon, characterizing it in terms of the window space required to support development tasks, and the overlapping of these windows. The level of chaos impacts on both the proportion of time that developers spend altering the UI of the IDE and the proportion of time spent performing program comprehension tasks. This corroborates the findings of the studies of Ko *et al.* [KMCA06] and Bragdon *et al.* [BRZ+10]. This chapter also discusses our first steps to help developers coping with the chaos. We devised and evaluated simple strategies that leverage IDE interactions to automatically reshape the UI of the IDE. Our findings reveal that simple strategies may considerably reduce space occupancy and time spent in a chaotic environment, making more time and space available for the real essence of software development.

Structure of the Chapter

Section 13.1 analyzes the literature and provides initial empirical evidence, through the analysis of Mylyn data, of the presence of chaos in the IDE UI. Section 13.2 details our primary dataset and explains how we modeled, characterized and measured the "chaos inside the IDE". In Section 13.3 we present our strategies to reduce the chaos present in the IDE and discuss their impact. Finally, Section 13.4 discusses the threats to validity of our work and concludes the chapter.

[1] Not to be confused with *"deterministic chaos"* [TS02].

13.1 IDEs and Chaotic UIs

The complexity of building and maintaining working sets for typical development tasks can both explain the chaotic configuration of an IDE, and be impacted by it.

The information path obtained from navigation in an information space reveals the user's mental model of the system [WM99]. In software engineering, developers spend a considerable portion of their time building and maintaining the working set of code fragments relevant to a task. This is challenging when the relevant code fragments are dispersed in several locations in the system. An observational study by Ko *et al.* reported that developers spend 35% of their time navigating the source code in search for information [KMCA06], and that 27% of the navigation operations are performed on already visited locations, indicating the necessity to periodically revisit these locations to recall information no longer visible on screen.

This study is not alone: The recent context model study by Fritz *et al.* [FBM+14], based on detailed observations of 12 developers, each solving three tasks, found that the average context model necessary to solve a task contained 4 classes. Further evidence is present in the study by Sillito *et al.* [SMDV08]: Developers ask a variety of questions during maintenance tasks; answering some of the questions involves inspecting several entities, increasing working set size.

13.1.1 Improving Management of Working Sets

Several approaches have been proposed to improve the management of working sets. Robillard and Murphy proposed to represent scattered concerns in source code as concern graphs [RM07]. Mylyn itself is such a tool, which monitors interaction data to automatically build a degree-of-interest model (DOI), altering the views of the *Eclipse* IDE by filtering out entities with a low DOI value [KM05]. Its effectiveness has been empirically demonstrated [KM06]. The Degree-of-Knowledge (DOK) model by Fritz *et al.* is an extension of the DOI, that also includes authorship [FSK+14]. Other tools monitoring interactions to help software exploration have been proposed, such as Navtracks [SES05], and TeamTracks [DCR05].

The Problem with Tab-Based UIs

A key issue not addressed by these works is the fact that most IDEs do not properly support the work of maintaining one's mental model by adopting a file-based representation of source code, while most working sets span several files. Moreover, the particular UI paradigm typically used to manipulate source code hinders the maintenance of complex working sets.

Eclipse is a good representative of widely used IDEs (such as *Visual Studio*, *Netbeans*, *IntelliJ Idea*) that adopt the tab-based metaphor. Each file is shown as an editor in the IDE, with navigation tools (Package explorers, search tools, *etc.*) shown as views around the central editor. By default, the screen estate allows for at most one file to be visible at the same time, while other open files are shown as "hidden" tabs. This is the case of the overwhelming majority of the *Java* developers broadcasting their coding sessions on the "livecoding" website[2]: Out of several dozens of videos linked on the site, only a handful of developers stray away from the IDE's default settings and use two code tabs at the same time, even if the vast majority of published coding videos show IDEs with several tabs open at the same time.

The study by Ko *et al.* [KMCA06] pointed out a considerable number of re-navigation to entities recently browsed (27%). It highlights patterns of *back-and-forth navigation* between two files to compare similar pieces of code, which is necessary if only one tab is visible at a time.

[2] See `https://www.livecoding.tv/videos/java/`

In a series of controlled experiments investigating the influence of type systems [HKR+14], and of API documentation [EHRS14, PHR14], researchers observed that the treatments with higher code completion times also had larger working sets, and a larger number of tab switches in a tab-based IDE. All of these findings highlight the issues related to dealing with multiple tabs in an IDE when several scattered fragments need to be accessed at the same time.

13.1.2 Evidence from Mylyn data

To explore in a more systematic way the phenomenon of how tab-based IDEs support the management of complex working sets, we measured the size of the task contexts[3] contained in the Mylyn dataset available in the *Eclipse* Bugzilla repository[4]. We downloaded 6,182 bug reports that had a Mylyn task context as attachment. For each task context, we counted the number of distinct *Java* files and methods that were interacted with. We applied filtering techniques to bypass some of the deficiencies of the data [SRG15], namely removing massive selection events that could lead to an overestimation of the number of entities interacted with (*e.g.,* selecting an entire group of classes from the navigation panel, without actually opening them). In essence, we filter events that originate less than 100ms after the previous event.

We are left with the number of *Java* files and the number of methods that were interacted with during a task (we exclude other types of files, such as .class or XML files). The median task context has interactions with 3 *Java* files, that is, at least half of the tasks involve interactions with at least three *Java* files. The upper quartile is 8, meaning that for at least 25% of the tasks, the developer needed to consult 8 or more java source code files to finish the task. Clearly, a large number of tasks demand non-trivial interactions with a large number of source code entities. Outliers are even higher; if we focus on methods, the median number of methods interacted with is 5, while the upper quartile is 21. This indicates that for at least 25% of the tasks, the developer had to piece together information from a large number of methods.

This is a likely sign that the typical size of working sets, together with the way that source code is represented by the tab-based UI paradigm, may generate chaos in the IDE, forcing developers to spend considerable time in interacting with the UI components to manipulate their working sets, for instance to revisit entities as documented by Ko [KMCA06].

Obtaining further evidence is hard, because of the Mylyn data itself, which presents several limitations for this particular investigation: It does not contain information on the visibility of elements on screen, so it is impossible to know how many tabs were open at distinct points during a task, or if developers had several tabs visible at the same time. There is no information to estimate the size of the screen, or the size of tabs. The data is aggregated, and it is not always possible to know the exact sequence of events (a sequence of events concerning an entity may be encoded as a time period where several events occured, reducing precision, as documented by Ying and Robillard [YR11]). Finally, the Mylyn data is based on a files-and-tabs metaphor, which is in itself limiting in terms of possible optimizations.

13.1.3 Beyond Tab-based IDEs

Recent efforts have investigated better program representations and UI paradigms than the file-and-tab-based metaphor of most common IDEs. We can trace back this inspiration to the *Lisp* and *Smalltalk* IDEs of the 80's, whose most recent representative is *Pharo*. Efforts include Code Canvas [DR10] and Code Bubbles [BZR+10]. Code Canvas has seen parts of its functionality

[3] A set of artifacts that Mylyn considers relevant for the task-at-hand.

[4] See `https://bugs.eclipse.org`

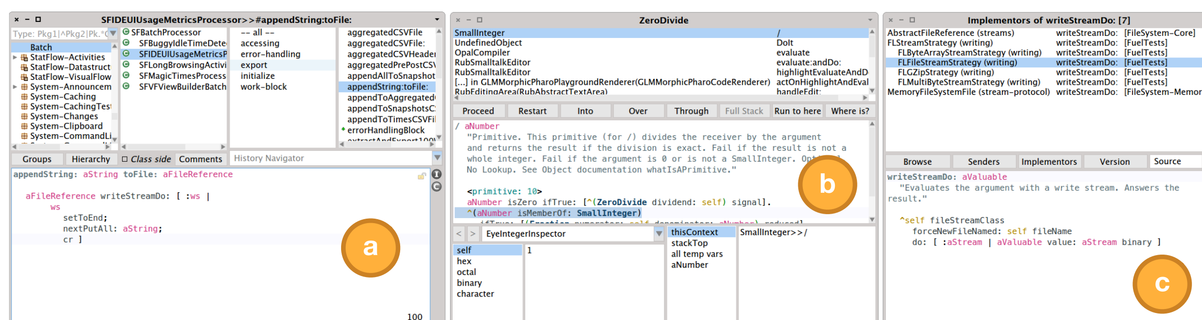


Figure 13.1. Main UIs to display code: (a) Browsers, (b) Debuggers, and (c) Message Lists

released in *Visual Studio* as DEBUGGER CANVAS, which also integrates parts of CODE BUBBLES's functionality [DBR+12]. These tools aim to reduce the amount of code navigation, by maximizing the number of entities visible at the same time.

The evaluation of CODE BUBBLES is particularly instructive. The authors showed that at a similar screen resolution, CODE BUBBLES was able to show more methods at the same time than the classic *Eclipse* view [BZR+10]. Furthermore, a controlled experiment showed that Code Bubbles users were both more successful and faster in completing maintenance tasks than *Eclipse* users [BRZ+10]. Parts of this performance increase is attributable to a reduction of repeated navigations, such as the ones observed by Ko, according to the videos recorded during the controlled experiment (75.9% of all *Eclipse* navigation operations, compared to 37.6% for CODE BUBBLES), as more entities were visible on screen.

13.1.4 Strengthening the existing evidence

Approaches such as CODE BUBBLES, and the insights that one can obtain from its evaluation, motivate the need to evaluate the impact of IDE UIs in alternative metaphors to the classic tab-based approach. Moreover, window-based IDEs also suffer from UI-related phenomena like the *window plague* [RND09, MML15c], and our work lies in the same area of research. We leverage our experience in recording and mining interaction data in the IDE [MML15b] to model and characterize the impact of chaos in window-based IDEs, evaluate possible techniques that can ameliorate the developer experience, and ultimately improve the support that UI components give in constructing and maintaining the working set by managing in a more efficient way the screen real estate. While the empirical evidence brought forth by CODE BUBBLES [BRZ+10] shows that increasing the number of code fragment visible at the same time has a positive impact on productivity, it is lacking in several aspects. Beyond the simple focus on the more general setting of window-based IDEs like *Pharo*, our study complements it in several ways.

Variety of Tasks. CODE BUBBLES's productivity benefits are shown in the context of a controlled experiment, with strong internal, but limited external validity [SSA15]. In fact, the study was conducted on two well-defined development tasks only, while both our datasets do not impose any constraint on the tasks at hand. This increases the external validity of the findings, at the price of a lower internal validity, *i.e.,* lower control with respect to an experimental setting.

Duration of Recorded Data. The conclusions in the CODE BUBBLES experiment are drawn from around 30 hours of development. On the other hand, our DFLOW dataset contains more than an order of magnitude of data (around 750 hours).

Entities Displayed. CODE BUBBLES was also evaluated on the number of methods shown on screen [BZR+10]. This was however done on a limited number of methods.

Impact of UI Components. We characterize the impact of chaos on the time spent on UI fiddling (*e.g.,* resizing windows) and the time spent on program understanding.

Recorded Interaction Data. Last but not least, our approach records enough interaction data to let us simulate the impact of various strategies on the chaos in the IDE, without needing to implement a prototype in the early stages.

These characteristics make our evaluation complementary to previous evaluations.

13.2 Charactering and Measuring the Chaos

Each development session is a self-contained and focused development period without long interruptions. This removes the potential problem of considering major interruptions (*e.g.,* Skype calls, coffee breaks) as part as the development flow. A development session is a sequence of IDE interactions captured with DFLOW which satisfies the followind constraints: i) All events happen in the same development environment/context (*i.e.,* an *image* in the Smalltalk jargon); ii) There are no adjacent pairs of events such that there are more than 5 minutes of inactivity between them; iii) When the user closes the IDE, the session terminates.

13.2.1 DFlow Dataset

Table 13.1 summarizes our dataset. It counts 771 hours of development time coming from 17 open-source and academic developers working on or around the *Pharo* project.

Table 13.1. Dataset – DFlow dataset to characterize and measure chaos

Metric		Value
Number of Sessions		1,631
Number of Developers		17
Development Time		771h 10m 21s
Avg. Session Duration		28m 22s
Metric	**Total**	**Avg. per Session**
Number of Windows	40,140	24.61
Number of Browsers	6,833	4.19
Number of Debuggers	2,844	1.74
Number of Message Lists	3,870	2.37
UI Time	102h 15m 30s	3m 55s
Understanding Time	594h 54m 57s	22m 49s

We collected interactions with more than 40,000 windows, that we further refined according to their type. We only consider interactions with windows whose aim is to display and let the user interact with source code:

- **Code Browsers** are the core windows to navigate, read, and write code (see Figure 13.1.a).

- **Debuggers** are the windows dedicated to debugging activities. They let users navigate the call stack, watch the state of variables, and read/edit in place the source code of a method (see Figure 13.1.b).
- **Message Lists** are all the UIs that display a list of methods and, upon selection, the source code of the method itself. Example includes UIs to browse implementors of a method or methods that invoke another method (see Figure 13.1.c).[5]

Interaction with Windows

We collected a total of 6,833 code browsers, 2,844 debuggers, and 3,870 message lists. Each session, on average, lasts for ca. half an hour and counts interactions with 24 generic windows. Considering only windows containing source code, on average each session features 4 browsers, 2 message lists, and 2 debuggers. However, these aggregate metrics are highly variable, with a considerable number of outliers. For example, considering sessions with windows containing code, the number of outliers (containing more than 18 of such windows) is 175, roughly corresponding to a tenth of the recorded sessions.

Activity Durations

In Chapter 7 we used interaction data to measure the time spent in several programming tasks, like editing, navigating and searching for code artifacts, interacting with the UI of the IDE, and performing corollary activities, such as object inspection at runtime [MML15b]. Two of these components are useful for our current study, when it comes to understand how the "level of chaos" correlates with the behavior of the user.

The first is the *UI Time*, devoted to fiddling with the UI of the IDE, *i.e.,* moving and resizing windows. Our dataset features more than 102 hours of UI Time (on average, ca. 3m 55s per session, ca. 14% of the total time).

The second is an estimate of the time devoted to program understanding. In our model [MML15b], we consider as understanding the sum of three components: i) basic understanding time; ii) time spent inspecting objects at runtime; and iii) time spent doing mouse drifting, *i.e.,* the time the user "drifts" with the mouse without clicking, for example to support code reading. Essentially, the basic understanding time is composed of all the time intervals without any recorded event in the interaction data that are greater than a given reaction time[6] (that in our model is equal to 1 second).

The reaction time models the time that elapses between the end of a physical action sequence, *i.e.,* typing or moving the mouse, and the beginning of concrete mental processes like reflecting, thinking, planning, etc. which represent the basic moments of program understanding. In total, we estimate more than 594 hours of understanding time, on average more than 22 minutes per session (*i.e.,* around 80% of the session duration).

13.2.2 Modeling Chaos

To characterize and quantify the "level of chaos" of a programming session in the IDE, we introduce a set of UI metrics related to the usage of the screen and we observe how they evolve throughout the sessions.

[5] Opening the call hierarchy of a method, in the *Eclipse* IDE.

[6] An approximation of the *Psychological Refractory Period* that varies between 0.15 and 1.5 seconds depending on the task [Pin99]. Please refer to Chapter 7 for our model of program understanding.

Table 13.2. Space Occupancy Metrics

Occupied Space	*The sum of the areas of all the screen regions occupied by 1+ windows*
Free Space	*The sum of the areas of all the screen regions* **not** *occupied by any window*
Focus Space	*The area of the screen region occupied by the active window*
Needed Space	*The total sum of the areas of all the windows,* i.e., *the space needed to display all windows*
Overlapping Space	*The sum of the areas of all the screen regions occupied by 2+ windows*
Overlapping Depth	*The number of overlapping windows in a given screen region*
Weighted Overlapping Space	*The sum of the areas of all the screen regions weighted by their overlapping depth*

Quantifying Chaos

We use the metrics described in Table 13.2 to measure how developers exploit the screen space.

Measuring Chaos of a Single Layout. Consider a given moment during a development session, with a fixed layout of windows in the screen. All the metrics listed in Table 13.2 rely on the concept of *screen region*, a part of the screen obtained by creating a grid on the screen using all the coordinates (*i.e.,* position) of the visible IDE windows in the screen in a given *snapshot* of a development session, see Figure 13.2 (right).

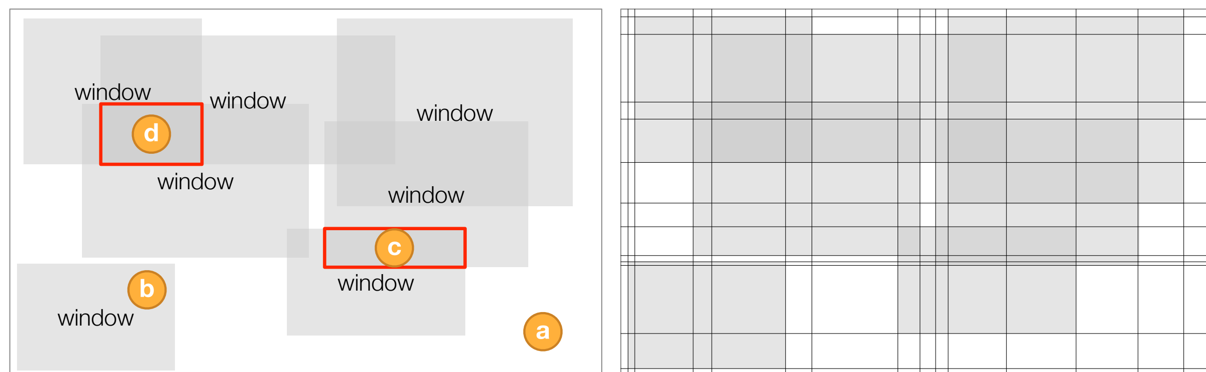


Figure 13.2. Visualizing a snapshot of a session (left) and the corresponding screen regions used to measure the chaos (right)

Figure 13.2 shows a visualization of a snapshot of a session (left) and its decomposition in screen regions (right). In the view, each window is depicted with a translucent gray rectangle (*i.e.,* with size and position proportional to the real window in the IDE). The white rectangle containing all windows represents the main IDE window.

In Figure 13.2 the darker the screen region, the more the overlapping between windows. The figure highlights different areas of the screen: (a) one with no windows (*i.e.,* free space), (b) one with a single window (*i.e.,* no overlapping), (c) one with low overlapping (*i.e.,* only 2 windows overlap), and (d) one with high overlapping (*i.e.,* 3+ windows overlap).

We quantify overlapping in three different ways: *overlapping space*, *overlapping depth*, and *weighted overlapping space*. The first measures the linear overlapping space expressed as the sum of the areas of all the screen regions occupied by more than two windows. The depth indicates, for each screen region, how many windows overlap. The weighted overlapping is a combination of the previous two measures that assigns more weight to regions with higher overlapping depth.

Aggregating Layout Changes. To observe the evolution of the chaos, we divide each session into *snapshots*. Figure 13.3 schematizes a development session, *i.e.,* a timeline of events. Among these events, there are events that induce a change of the layout of the IDE (depicted in red) and events that do not change the layout of the IDE (depicted in gray). The former are, for example, when the user resizes a window, opens a new window, or closes an existing window.

Figure 13.3. Session snapshots explained

We call *snapshot* each sequence of events that begins with a layout-changing event and continues until the last event before the next layout-changing event (with the exception of the first and last snapshot that are delimited by the session start and session end respectively). In short, a snapshot is a period in which the layout of the IDE is fixed, thus the values of all the metrics listed in Table 13.2 are constant for the entire duration of the snapshot. In the example depicted in Figure 13.3, there are four snapshots, from S#1 to S#4.

To aggregate and define the level of chaos of a session, we average the values of metrics for each snapshot. Table 13.3 summarizes the distribution of space occupancy metrics across all sessions. For each metric we report the average and quartile values across all the sessions. Values (except for the overlapping depth) are expressed in percentage with respect to the available screen space (*i.e.,* resolution).

Table 13.3. Results – Distribution of Space Occupancy Metrics across all sessions

		Quartiles		
Name	Avg.	Q_1	Q_2	Q_3
Occupied	48.22%	36.16%	43.78%	61.37%
Free	51.78%	38.63%	56.22%	63.84%
Focus	32.66%	20.06%	35.22%	40.12%
Needed	96.83%	39.35%	73.47%	119.53%
Overlapping	20.95%	0.00%	17.08%	35.97%
Weighted Overlapping	69.05%	0.00%	39.61%	95.59%
Overlapping Depth	2.76	1.00	2.00	3.40

On average, the needed space is around 100%, which seems to imply that developers do not experience chaos. However, the average occupied space is around 50% of the screen, with 21% overlapping, indicating at least a need for a better management of the screen real estate. In 25% of the sessions (*i.e.,* Q1) windows do not overlap at all, while in the last quartile the average needed space is always above 120%.

To better analyze the impact of chaos in the recorded development sessions and shed light into how chaos impacts development activities, we group the metric values into categories and we take in consideration the time dimension.

Categories of Chaos

To characterize the level of chaos in a session we use the space occupancy metrics. The most important indicators are the needed space and the overlapping space. However, these values are *continuous*, and it is not clear at which thresholds of values the chaos is acceptable or not.

Having a high needed space (*e.g.,* ≥100% of the available space) means that there is no way a developer can re-arrange the open windows to fit them all on screen. This forces her to have overlapping windows, which is suboptimal for an efficient working environment. In fact, this intuitive correlation holds across our entire dataset: Needed space and weighted overlapping have a strong positive correlation with the *Pearson Correlation Coefficient* PCC=0.99 (statistically significant at 95% confidence interval with p-value 2.2E-16). Thus, we define chaos levels using only one indicator, the needed space, because it is a more intuitive metric than a weighted sum.

We identify two macro-levels of chaos: low- and high-chaos. The threshold that distinguishes between the two levels is 100%, *i.e.,* when the screen resolution is, ideally, enough to accomodate all the open windows, we say that the chaos is low. We use the term "ideally" because the needed space does not take into account overlapping, *i.e.,* having less than 100% of needed space does not imply that windows are uniformly distributed in the screen without overlapping. Conversely, if the screen resolution is insufficient (needed space >100%) we say that the chaos is high. We refined the two macro-levels into four levels: *Comfy*, *Ok*, *Mess*, and *Hell*, detailed in Table 13.4.

Table 13.4. Chaos-Levels: Comfy, Ok, Mess, and Hell

Comfy	*≤75% of the screen is required to layout windows. The user can still manipulate and rearrange windows in a comfortable manner, supporting the task at hand, which requires a likely small/reduced working set.*
Ok	*>75% and ≤100% of screen is required to layout all windows*
Mess	*>100% and ≤200% of screen is required to layout all windows*
Hell	*>200% of screen is required to layout all the windows,* i.e., *a developer would need more than two screens (needed space >200%) to arrange all the currently opened windows.*

Justifying the Thresholds. To define the levels of chaos we chose three thresholds for the value of the needed space metric: 75%, 100%, and 200%. Defining these thresholds in a systematic and objective way it is far from trivial, if not impossible. The perceived level of chaos is very subjective and depends on several factors, *e.g.,* resolution, screen size. As a test we computed the values that better divide our data in four partitions (as the number of levels of chaos) using a k-means clustering algorithm [Mac67]. The thresholds are respectively 60%, 110%, and 190%. The threshold for *Comfy* is imprecise, however, our choice is more conservative.

Time spent in Chaos Configurations. Table 13.5 summarizes the average values (per session) of the time spent in each of the four chaos-levels. For each level we report the percentage of the time spent (with respect to the total duration of each session) and the absolute value. All values are averages across all the sessions in our dataset.

Table 13.5. Results – Average time spent per chaos-level

	%	Duration
Comfy	51.04%	10m 50s 126ms
Ok	16.98%	5m 10s 633ms
Mess	21.11%	7m 15s 16ms
Hell	10.88%	5m 26s 922ms

The results above show that for around 32% of the time developers work in a high-chaos setting, *i.e.,* for a session of around 30 minutes of work, more than 12 minutes are spent working with windows occupying more than 100% of the screen. Moreover, 5 out of 30 minutes are

spent in a more critical setting, the hell configuration, where the needed space is over 200% the resolution. The total time spent in high-chaos amounts to ca. 331 hours, *i.e.,* 42.92% of development time.

How Does Chaos Correlate with the UI Time and Program Understanding? For each session, we correlate the percentage of time spent in each configuration with the time spend in fiddling with the UI of the IDE and the understanding time, using the test for PCC. Table 13.6 shows the results of these tests.

Table 13.6. Results – Chaos, UI, and Understanding Time

	UI		**Understanding**	
	PCC	*p-value*	*PCC*	*p-value*
Comfy	-0.34	2.20E-16	-0.27	2.20E-16
Ok	-0.04	1.03E-01	0.05	3.36E-02
Mess	0.16	4.42E-10	0.11	1.940E-05
Hell	0.42	2.20E-16	0.26	2.20E-16

The moderate correlations are expected when considering that understanding time, for example, is influenced by a multitude of factors, including not only the size of working sets but the quality of code or the difficulty of the task at hand.

On high-chaos levels, developers likely spend more time fiddling with the UI and on program understanding. This is consistent with a likely presence of more complex working sets, spread on multiple windows, that require more attention and time to be managed. The correlation between the time spent on the hell configuration and the UI time is particularly strong (0.42 PCC, p-value 2.20E-16).

On the comfy configuration, there is statistically significant evidence of moderate negative correlation on both UI time and understanding time. This is consistent with the fact that smaller working set support likely more "productive" sessions, where less time is spent on managing the UI and where mental processes are more effective. There is no evidence of correlations in the ok level of chaos. Probably, on the typical configurations of windows corresponding to the thresholds of needed space of this category, other factors prevail.

13.2.3 Wrapping Up

We modeled chaos in window-based IDEs by considering the needed space to visualize all code containing windows without overlapping. We defined four categories of chaos, and by leveraging more than 770 hours of interaction data, we showed that developers spend more than 30% of their time in a high-chaos configuration, corroborating previous research along the same lines.

We also discussed how our data is also consistent with potential impact to the time spent by developers in program understanding and UI time. In the following section, we simulate and discuss how even simple elision strategies and automatic window layouts can improve the level of chaos experienced by developers.

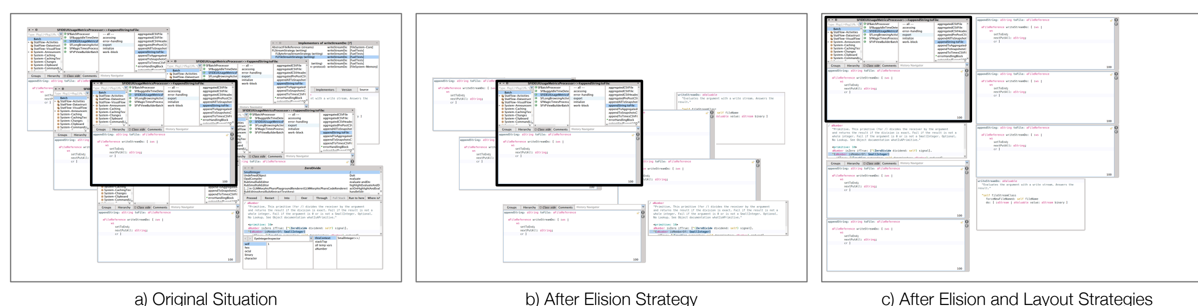
13.3 Make Code, not Chaos

The previous section provided evidence that developers have to cope with chaotic environments during a third of their programming time, with negative implications both in terms of time spent fiddling with the UI, as well as additional time spent with program understanding. This section

explains how we can tame the IDE, by adopting simple mechanisms to reduce the amount of needed space on the screen.

13.3.1 Strategies to Tame the UI of the IDE

Figure 13.4 exemplifies our two strategies: Elision and Layout. The strategies are part of a two-step process: We first reclaim space by eliding (hiding) the redundant parts of each non-active window (in the figure, the in-focus or active window is depicted with a thick border). Then, we apply a new layout to occupy the space more efficiently.



a) Original Situation   b) After Elision Strategy   c) After Elision and Layout Strategies

Figure 13.4. Elision and Layout Strategies in a nutshell

In the example depicted in Figure 13.4, there is not enough space to position all the windows of the IDE; consequently, the overlapping between the windows is relatively high (see Figure 13.4.a). After the application of the elision strategy (see Figure 13.4.b) the free space in the IDE increases, but the overlapping is still present. Finally, with the new layout, all the (elided) windows are now positioned in the IDE without overlapping (see Figure 13.4.c).

Elision Strategy

The elision strategy hides part of a window to reduce the visual cognitive load on the developer. It stems from the observation that at each instant there is only one active window; all the others are inactive, producing a considerable amount of visual noise. The underlying idea is to leave the active window untouched, while reducing the visual noise present in the background windows. The goal is to *keep the code displayed in all windows visible* while hiding the non-code elements displayed in the window (lists, buttons, *etc.*). These UI elements are mostly used for navigation, and are only usable while the window is active. When the focus changes, its elided elements are restored. Since different types of windows display source code in different ways, the strategy implementation depends on the window types.

Code Browsers and Message Lists display code in the bottom half of the window. The top half contains source code navigation elements. Our strategy elides the top part while keeping the code visible. Figures 13.5.a and 13.5.c illustrate how the strategy works on these cases, reducing the needed space of non-active windows by 50%.

Debuggers display the code of a method on the stack and let the user modify it. The source code pane is in the central part of the window (occupying roughly 1/3 of the window). Our strategy elides the top and the bottom parts while keeping the central part (*i.e.,* code) visible. Figure 13.5.b shows how the strategy works, reducing the space occupied by each non-active

Figure 13.5. Elision strategy for (a) Code Browsers, (b) Debuggers, and (c) Message Lists

debugger by ca. 66%. Figure 13.5 shows our elision strategy, applied on the same windows of Figure 13.1, reducing the opacity of the elided parts, instead of hiding them.

Layout Strategy

The elision strategy efficiently reduces the amount of needed space occupied by non-active windows. However, also the overlapping between windows contributes to chaos, *i.e.,* by hiding parts of the open windows that might be relevant for the developer. For the sake of simplicity our definition of chaos (see Table 13.4) only considers the needed space, however as discussed in Section 13.2.2 needed space and weighted overlapping have a very strong positive correlation. Thus, reducing the overlapping might contribute to the reduction of the chaos level.

To reduce the overlapping, we adopt a layout algorithm inspired by the rectangle packing layout. The idea is to stack all the windows in columns from the origin of the screen (*i.e.,* top-left) one below the other, as shown in Figure 13.4.c. If a window cannot be repositioned (*i.e.,* it does not fit in the screen if moved in the new position), it is left in the original place.

Wrapping Up

We discussed two strategies to tame the chaos in the IDE: Elision and Layout. Figure 13.4 summarizes, step-by-step, how these strategies work on a hypothetical development session. Intuitively, elision and layout strategies help to tame the chaos inside the IDE by reducing both the space needed to display all the windows and the overlapping between them. The elision strategy aims to reduce the amount of visual noise in the IDE while the layout strategy takes care of reducing the overlapping between windows. Next, we evaluate the impact of the strategies under different perspectives.

13.3.2 Impact of Elision and Layout Strategies

To determine the potential impact of the strategies, we simulate their application on our dataset of recorded sessions. For each snapshot, we applied both the elision strategy alone and together with the layout strategy. Then, we discuss how the strategies impact the occupied space, then how they impact the time spent in each chaos level.

On Space Occupancy. We compute the gain for the space occupancy metrics as percentages with respect to the baseline, *i.e.,* the metric value before applying the strategies, as follows:

$$Gain\ (\%) = \frac{Metric_{after} - Metric_{before}}{Metric_{before}}$$

Suppose that for a session, the value of the *free space* before applying the strategies (*i.e.,* the baseline) is 12.82%. If, after applying the strategy, the free space increases to 22.32%, the relative gain would be 74.10%.

Table 13.7 reports the average of differences before and after applying each strategy (*i.e.,* $Metric_{after} - Metric_{before}$), together with the corresponding average gain.

Table 13.7. Results – Percentage gain of Space Occupancy Metrics

	Elision		**Elision + Layout**	
Metric	Average Gain (%)	Average of Differences	Average Gain (%)	Average of Differences
Occupied Space	-13.44%	-7.28%	6.99%	1.38%
Free Space	27.82%	7.28%	4.82%	-1.38%
Needed Space	-24.74%	-32.09%	-24.74%	-32.09%
Overlapping Space	-34.61%	-9.61%	-54.68%	-13.53%
Weighted Overlapping	-36.64%	-34.30%	-58.13%	-47.60%
Overlapping Depth	-2.59%	-0.10	-26.43%	-0.93

The simple elision strategy is—as expected—able to significantly increase the amount of free space, by almost 28%. This is accompanied by a general improvement of all the occupancy metrics, *e.g.,* needed space drops by almost 25%. Moreover, overlapping space considerably drops (ca. 35%), even if this strategy does not try to consciously reduce it.

The effect of windows re-layout produces configurations which make better use of the screen real estate. After layout, the needed space does not obviously change (with the same relative decrease of about 25% due to elision), but the overlapping space is reduced by more than 50%. The more efficient layout better use of available space is visible on the free space metric, which drops considerably compared to elision alone. In addition, the occupied space actually increases compared to the default configuration. This is in line with the goal of distributed windows in a more space efficient configuration.

To verify whether the effects of strategies are significant, we performed a paired t-test between the values of each metric before and after applying the strategies. For all metrics but occupied and free space, we observe that the metric values are reduced with statistical significance with both strategies (confidence interval 95%, *p-value* $< 2.2 \cdot 10^{-16}$). The same happens for the occupied space metric after applying the elision strategy, and for the free space metric after elision and re-layout. Instead, for the free space metric value after applying just elision, and for the occupied space metric after applying elision and re-layout, we find a significant increase of the metric value (respectively *p-value* $< 2.2 \cdot 10^{-16}$ and $7.8 \cdot 10^{-8}$, again at 95% confidence interval).

On Chaos Time. Table 13.8 shows the impact of our strategies on the time spent in each of the chaos categories defined in our model. Since the categories are defined only in term of needed space, the results refer to either strategies.

The elision strategy has essentially the effect of redistributing the time spent on each category towards less chaotic categories. The time spent in the most chaotic category (*i.e.,* hell) is reduced on average around 8% of the session time. The second high-chaos category, mess, is reduced again by 8% on average for each session. These times are redistributed mostly towards the comfy category, which gains around 18% of average time in each session, while the ok category changes slightly. Developers spent 30% of their time in chaos previously; using the elision strategy could reduce this amount down to 14%, less than half.

Table 13.8. Results – Percentage gain and delta time

	Avg. Gain per Session (%)	*Absolute Difference*
Comfy	17.73%	137h 50m 44s 882ms
Ok	-1.35%	3h 25m 58s 502ms
Mess	-8.08%	-40h 29m 38s 554ms
Hell	-8.30%	-100h 47m 04s 812ms

Looking at the variation in the total amount of time spent per category, we find that programmers spent 142h in the hell category in original settings, while they spend 100h less in the new settings, a reduction of 70%. Time spent in the mess state drops from 188h to 148h (21%). The ok state is relatively stable, from 134h to 131h (-2%). The winner is the comfy category, which increases from 282h to 419h, a 48% increase.

Overall, the total recorded time spent in high-chaos categories amounts to 42.92%, while after elision this time would drop to 24.60%. In addition to get a better understanding of the improvement of the layout strategy, we compute the average drop in overlapping space for each of the four chaos levels.

Table 13.9. Results – Average Weighted Overlapping per chaos-level

	Original	*Elision*	*Elision + Layout*
Comfy	17.39%	16.76%	1.67%
Ok	55.86%	53.19%	27.81%
Mess	114.71%	112.99%	88.04%
Hell	330.11%	259.22%	235.52%

Table 13.9 shows the reductions of average weighted overlapping before and after each strategy. We find that after just elision, in almost all categories there is no large change in overlapping, except in the hell category where we see a 27% drop. A dual effect happens after laying out, where major effects happen in the comfy category (relative drop of 90%) and we see large drops in the ok and mess categories. The effect in the hell category is less pronounced. In addition to spending less time in high chaos levels, developers would additionally enjoy a better spatial organization, particularly in the comfy and ok categories.

13.3.3 Threats to Validity

Internal Validity. Our definition of chaos is based on overlapping and needed space. Potentially different developers might have additional indicators of chaos. To cope with this threat we plan to cross-validate our measures of chaos with concrete observations (*e.g.,* think-aloud) to better grasp the sensitivity of developers to chaos. Another threat concerns our layout strategy that messes up that spatial memory of developers. This naïve strategy is only a proof of concept that simple means can already achieve a lot. We are aware of the importance of user placement of windows [HC86, RvDR$^+$00] and in our future work we will devise strategies that consider and preserve the spatial memory of developers. Another threat is that we only simulated strategies on our existing dataset: We only replay the past interactions of the developers in our dataset; were the developers to use our elision and layout strategies, they might behave differently. We expect that as a result of using the elision and layout strategies, developers would spend less time UI fiddling and revisiting previous source code locations, as these would stay on screen.

To mitigate this threat, we performed a correlation study between the time spent by developers in fiddling with the UI of the IDE and the levels of chaos (see Table 13.6). We found evidence that UI and program comprehension time are positively correlated with the high-chaos levels and negatively correlated with low-chaos levels, supporting the fact that less chaotic environments might let developers spend less time in taming the IDE. We are of course aware that correlation is not causation – to obtain better results we should collect feedback and new measurements from developers in the field, similar to the experiment on CODE BUBBLES [BRZ+10].

Statistical Conclusion. We considered more than 770 hours of development affecting more than 40k windows. Our dataset supported us in drawing statistically significant conclusions about correlation between chaos in the IDE and both the time spent by developers altering the UI of the IDE, and the time spent performing program comprehension tasks.

External Validity. We focused on the *Pharo* IDE and the fine-grained interaction data we collected. Results may vary for different programming languages and IDEs. However, as part as our motivation (see Section 13.1) we extensively discussed the situation in tab-based IDEs (*e.g., Eclipse*) and provided preliminary evidence, leveraging MYLYN data, that also this UI paradigm may generate chaos in the IDE. Unfortunately, due to the coarse nature of MYLYN data, we could not conduct analyses at the same granularity of DFLOW. Bragdon's study—which had *Eclipse* as a baseline—gives confidence that our results would also be valid for other IDEs [BRZ+10]. Another threat is related to the distribution of recorded sessions among developers: Most of the sessions come from only 5 developers. This might influence conclusions about developer diversity, but this was not the focus of our work.

13.3.4 Wrapping up

With our strategies the time spent in high chaos can be considerably reduced, and indeed we could better manage the screen real estate. We do not evaluate how the reduction of time spent in high chaos level could impact the time spent in specific activities, but we have some confidence that this could indeed happen given the correlations we found in Table 13.6.

13.4 Reflections

The UIs offered to developers to browse complex relationships between source code are often inadequate. Thus, developers are repeatedly forced to use multiple UI components at the same time, bringing the IDE into a chaotic state. It is unclear to what extent chaos impacts development, and more importantly it is unclear how to tame it.

We analyzed a large dataset of fine-grained interaction data, counting more than 770 hours of development. We found that developers in our dataset spend more than 30% of their time in high levels of chaos. Furthermore, time spent in high levels of chaos is correlated with time spent fiddling with the UI. We proposed two simple strategies to reduce the chaos in the IDE and observed that they could save a considerable amount of space the IDE needs to layout windows.

One might argue that all these are mere user interface concerns, and not relevant for software engineering. However, while considerable efforts are spent in making mainstream end-user tools friendly, software developers are still using convoluted environments. We believe there is no good reason for why developers should be treated differently from "*normal*" users.

Part V

Epilogue

14

Long-Term Vision

In this dissertation we extensively discussed the importance of interaction data and examined different approaches to analyze and support developers inside the IDE. We believe that being aware of the interaction of developers is only the first step to kick off a new era for IDEs. In this chapter we discuss five possible research directions to further leverage the potential of interaction data inside the IDE.

Navtracks [SES05] and Teamtracks [DCR05] are tools that leverage a limited set of developer interactions with the IDE to support browsing through software. Extending this idea, we envision recommender systems that leverage a large set of fine-grained IDE interactions to support a variety of activities, such as debugging or the acquisition of crowd knowledge. In Part III we discussed a number of approaches to visually analyze interaction histories *a posteriori*. We believe that visualizations should be *"live"* and *"adaptive"*. Views should display interaction data inside the IDE *as they happen* and mutate their layout according to the context or the needs of the developer. Similarly, in the last 20 years the UIs of IDEs are largely unchanged being essentially "*glorified text editors*" that treat source code as text [Nie16]. We believe that these user interfaces should change. For example, UIs could be aware of the interactions of developers and adapt their shape by hiding part of their components, as we discussed in Chapter 13. The *Glamorous Toolkit*[1] developed by Chiş *et al.*, shares this vision and changes the UI of the *Pharo* IDE with "*moldable tools*", tools that can be easily extended to support various domain-specific abstraction and that are able to sense the domain model to use the correct extension [Chi16]. In this thesis we focused on IDE interaction data. However, besides IDEs, developers also interact with other tools. We envision the "*mother*" of all interaction profilers as an extensible suite of tools and plug-ins that monitors developer interactions with different tools. Finally, after collecting interactions of different developers with different sources, we foresee the development of the concept of crowdsourced holistic mental models, collaborative mental models that consider more than pure source code information.

Structure of the Chapter

Section 14.1 unveils the mother of all interaction profilers while Section 14.2 explains how fine-grained interaction data can support novel recommender systems. Section 14.3 discusses adaptive and live visualizations. In Section 14.4 we discuss about the future of the UIs of IDEs. Finally, Section 14.5 presents the idea of crowdsourced holistic mental models.

[1] See `http://gtoolkit.org`

14.1 Eye: The "Mother" of All Interaction Profilers

DFLOW silently observes the interactions of developers with the *Pharo* IDE. Chapter 7 explained how we can use IDE interactions to better understand what developers do inside the IDE. According to our results, developers spend about 10% of their work time outside the IDE [MML15b]. Development is an interleaving of web foraging, learning, and code writing [BGL+09].

For these reasons, we believe that to have a complete overview of what modern software development actually is, we should observe a larger set of tools and applications. To this aim we envision the *mother* of all interaction profilers that we call "EYE". More than a tool, this is an infrastructure that monitors how developers use the entire operating system at different levels. The core of EYE collects high-level information on how the developer uses the Operating System (OS), for example how much time the user spends on each different application, *i.e.*, IDE, Web Browser, Mail Client. EYE also offers an interface to domain-specific interaction profilers that gather detailed pieces of information on the usage of various tools. Figure 14.1 summarizes a potential architecture of such an infrastructure.

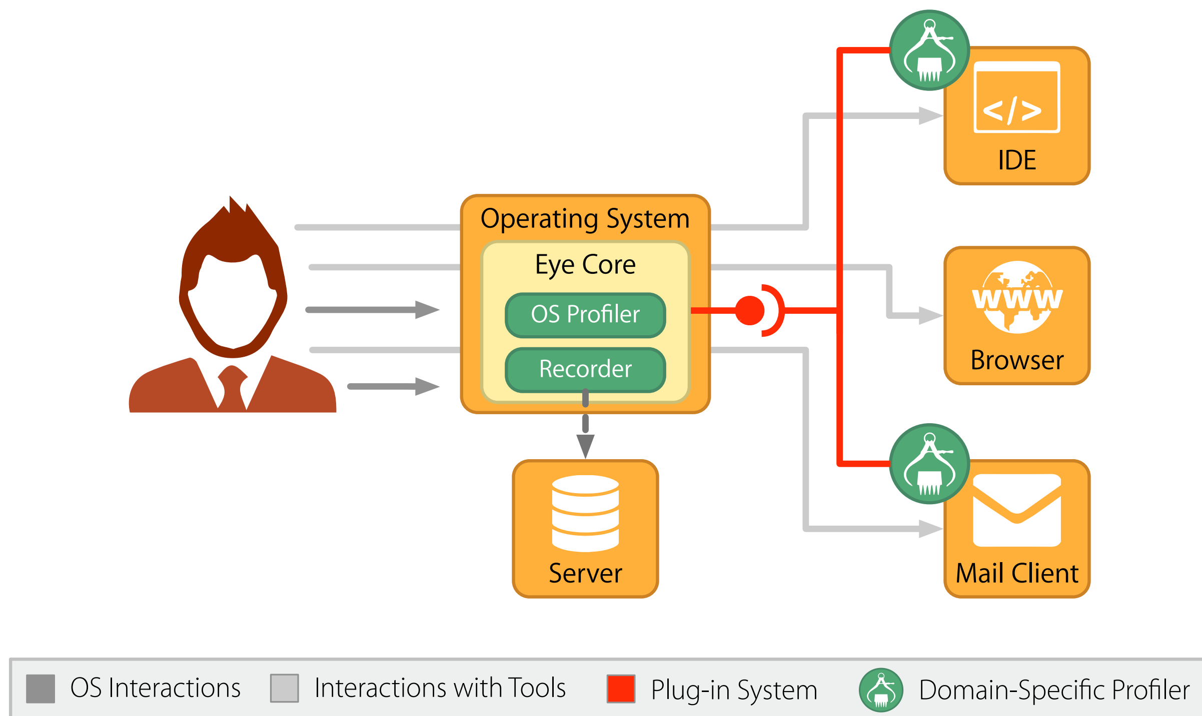


Figure 14.1. Potential architecture for the infrastructure of Eye

The developer always interacts with the OS. Some of her interactions are *pure* interactions with the OS while others *pass-through* the OS to reach specific tools. In the example of Figure 14.1, the developer interacts twice directly with the OS (*i.e.*, dark gray arrows) while three of her interactions (*i.e.*, light gray arrows) reach specific tools: IDE, Web Browser, and Mail Client. Basic meta-data about all these interactions are captured at OS-level by the basic *OS Profiler* that is part of the core of EYE. The configuration in this example, however, also includes two *domain-specific* profilers, one for the IDE and one for the Mail Client. These profilers gather more detailed data on the usage of these tools and report the collected data to the core components of EYE so the it can be recorded and sent to the centralized data collection server.

In the configuration of Figure 14.1, there is no domain-specific profiler for the Web Browser.

The core of EYE, however, gathers some basic pieces of information on its usage such as, for example, the time spent in the browser and the number of tabs and windows opened.

14.1.1 A Suite of Domain-Specific Interaction Profilers

Developers can extend the EYE infrastructure by adding their own domain-specific interaction profilers to gather more precise information about the use of specific tools during development. Below a non exhaustive list of 7 possible alternatives.

IDE Profiler. An IDE profiler, such as DFLOW, observes and records all the fine-grained IDE interactions inside the IDE, making them available for further use;

Fine-grained Source Code Changes Profiler. We believe that there is a relationship between interaction data and source code. In particular, it is likely that the more the code is complicated, the more the developer struggles with the IDE, *i.e.,* more navigations, more understanding time, less code written. For this reason, we should enrich plain IDE interaction data with fine-grained change data, such as the ones collected with SPYWARE [RL08]. To understand if there is some relation, we could devise metrics based on interaction data and correlate them with traditional source code metrics [HS95];

Operating System Interaction Profiler. DFLOW observes interactions inside the IDE. However, developers also use the operating system to perform actions somewhat connected to development, *e.g.,* moving or deleting source code files in the file system, interacting with its UIs, or copying and pasting snippets of code;

CLI Profiler. Nowadays most of the development happens inside IDEs. However, there are various activities that developers prefer to perform using a command-line interface (CLI) such as the UNIX shell;

Text Editor Profiler. Developers might also use text editors to write small snippets of code. A Text Editor Profiler observes when and how developers prefer this tool instead of the IDE;

Web Browser Profiler. Developers spend quite some time outside the IDE, for example foraging information in the web [BGL+09]. A Web Browser Profiler observes and records how the developers uses the Web Browser, *e.g.,* which pages she visits, how long she stays on the page, which hyperlinks she follows, *etc.*

Mail Client Profiler. It observes and records how the developer uses the Mail Client, *i.e.,* which e-mails she sends or reads, *etc.* E-mails can be parsed and linked to source code elements by using, for example, island grammars and island parsing techniques [Moo01]. Afterwards, we can augment the Mail Client with hyperlinks that enable the user to directly jump from the Mail Client to the correct source code location in the IDE.

Multiple Instances of Domain-Specific Profilers

EYE enables the user to install multiple instances of each *type* of domain-specific profiler. For example, one might be interested in having general time statistics of how the developer uses the Web Browser and more detailed information about the use of particular websites, such as *StackOverflow* (*i.e.,* by using black- or white-lists of allowed or blocked websites). In this case she can implement two different *Web Browser Profilers* and plug them in the EYE architecture.

14.1.2 All that Glitters Ain't Gold

Collecting potentially sensible data from human subjects poses a number of privacy and ethics concerns that we already discussed in the prologue of our dissertation (see Section 4.4). EYE pushes these issues to the limit. Despite the fact that EYE could provide developers with unprecedented support for their complete workflow, developers willingness to use EYE have to pay a very high price in terms of security and privacy.

The implementation of EYE should deeply consider privacy concerns and adopt severe measures to limit security threats and vulnerabilities of the infrastructure.

14.2 Recommender Systems Based on IDE Interactions

Software engineering is an activity that forces developers to deal with a huge information space: Thousands of lines of codes, hidden relationships between artifacts, and concerns scattered across the code base. Researchers invented tools, called *recommender (or recommendation) systems*, that support users in their decision-making while interacting with large information spaces.[2] These tools reduce the problem of information overload by exposing the user with the most interesting pieces of information considered relevant for the task at hand. Robillard *et al.* tailored this concept to the field of software engineering with *Recommendation Systems for Software Engineering (RSSEs)*, tools that provide information items that are estimated to be valuable for a software engineering task in a given context [RWZ10]. One of the most challenging aspects of RSSEs is indeed establishing the context to decide which items are potentially relevant [RWZ10].

As we discussed previously, IDE interactions model the behavior of developers inside the IDE. It has been shown that this information can be used to identify the development context for the task at hand [KM05, TFMH10, MML16b]. For this reason, we believe that this data should be used in the context of RSSEs to better identify the relevant information about the user, her environment, and the task at hand at the time of the recommendation. NAVTRACKS [SES05] and TEAMTRACKS [DCR05] are two examples of tools that leverage a limited set of IDE interactions aimed at supporting browsing through software. Bringing this idea forward, we believe that future recommenders should leverage a bigger set of more fine-grained IDE interactions, as the ones collected by DFLOW. Moreover, besides navigation, tools should support a broader set of activities, such as debugging. Debugging is among the most difficult activities performed by developers that can occupy a significant amount of their time [Bro85]. By leveraging previous fine-grained debugging histories, an RSSE could support developers to automate recurring "*debugging patterns*". In the context of *Pharo*, for example, debugging the *announcements* framework is very complicated [CNG13]. In their work, Chiş *et al.* explained the difficulties in debugging announcements and proposed a specific tool only for debugging this framework. Chiş also told us that some of these difficulties can be mitigated by suggesting developers some debugging steps that are very recurring in this particular context.

Moreover, on top of the vision discussed in Section 14.1, we could build RSSEs leveraging information coming from different sources such as IDEs, web-browsers, mail clients, *etc.* Recently, Ponzanelli *et al.* developed LIBRA, the first holistic recommender system [PSB+17]. LIBRA gathers information from both the IDE and the web browser to offer better support to information navigation and retrieval during development. Combining their approach with fine-grained interactions coming from different sources, we could build more complex holistic RSSEs.

[2] Definition introduced at RecSys 2009 (*ACM International Conference on Recommender Systems).*

According to Robillard *et al.*, RSSEs of the future should be proactive rather than passive: Instead of waiting for developers to invoke them, the RSSE should automatically deliver the right recommendation when needed [RWZ10]. Fine-grained IDE interaction data can also support this proactiveness by identifying, for example, when the developer is struggling to find a piece of information *i.e.,* inactivity or moving in circles between the same program entities.

14.3 Live and Adaptive Visualizations

Part III discussed a number of approaches to visually analyze interaction histories. All the proposed visualizations share a limitation: They only enable to visualize the data *a posteriori*. We believe that visualizations should be *"live"* and *"adaptive"*. Live visualizations display interaction data inside the IDE *as they happen*. Adaptive views are able to mutate their layout according to the context or the needs of the developer.

Live Visualizations. A live view co-evolves with the software system and always depicts its current state. An example is to depict user interactions in a tree-map layout.[3] Figure 14.2 shows the entire *Pharo* system and emphasizes (*i.e.,* using colors and partial expansion of nodes) the program entities involved in the development session. Nodes are nested using structural properties of code (*e.g.,* inheritance) and their size is proportional to software metrics. In the current implementation, developers can trigger the view *a posteriori*. However, this view could become "live" and be always open during development to emphasize the most recent activities acting as a *"visual memory"* for developers. While working a developer will, unconsciously, associate slices of her session with parts of the view. Later, she might better remember the part of the view with respect to the part of the code and use the view as a navigation means.

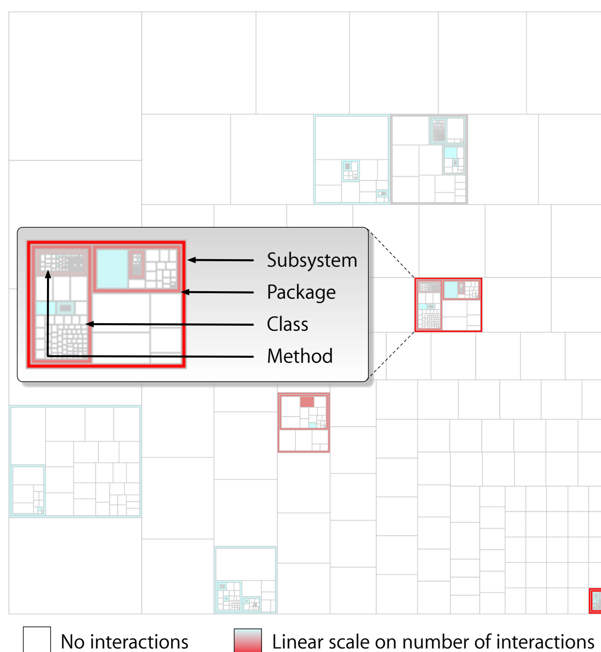


Figure 14.2. Treemap of IDE interactions

Adaptive Visualizations. In addition to be live, visualizations should be adaptive. An adaptive visualization is able, depending on the context, the history, and the task at hand, to reshape itself. We should prepare a catalogue of visualization to present interaction data from different perspectives. The IDE should be trained to assess the most beneficial visualization for the task at hand. At the beginning, for example, it can show a tree-map of interactions. As development advances, the IDE assesses whether there is an alternative presentation of the data that can potentially be more useful for the developer and adapt it accordingly, *i.e.,* changing the layout, the color scheme, or the zoom-level.

[3] We implemented the *"Squarified Treemaps"* algorithm of by Bruls *et al.* [BHvW99]. Our implementation is included in ROASSAL, the visualization engine of the *Pharo* IDE [BCDL13].

14.4 Adaptive User Interfaces

Developers spend much of their time reading and analyzing code, but mainstream IDEs are essentially glorified text editors mostly treating source code as text [Nie16].

> *"Although developers are known to spend much of their development time reading and analyzing code, mainstream IDEs do not do a good job of supporting program comprehension.*
>
> *IDEs are basically glorified text editors."*
>
> — Oscar Nierstrasz [Nie16]

As support of this thesis, Figure 14.3 shows user interface of the *Smalltalk-80* IDE (1983) compared with the *Pharo* IDE (2017) which are essentially unchanged in the last 30 years. Similarly, Figure 14.4 shows user interface of *Eclipse 1.0* (2001) compared with the one of its last release, *Eclipse Oxygen* (2017).

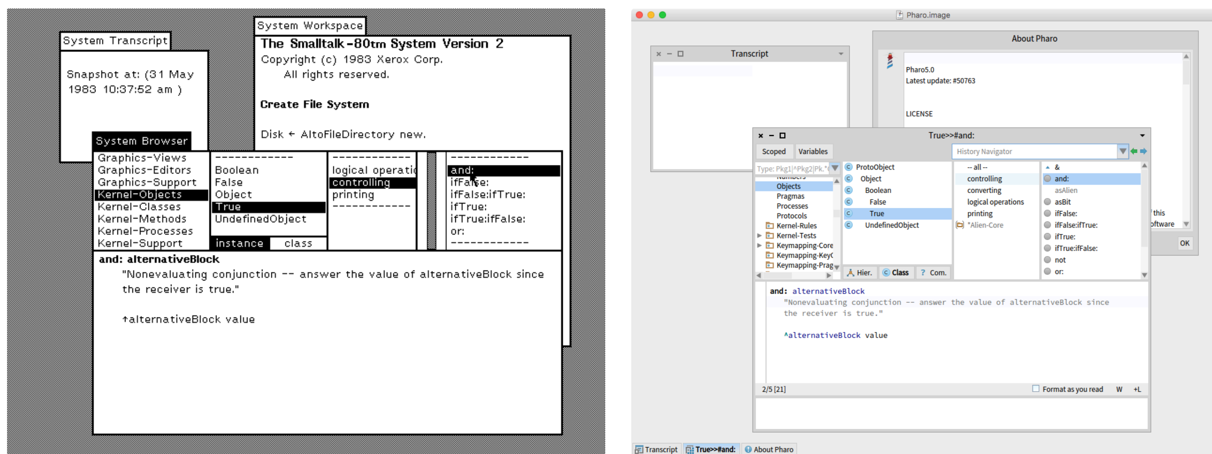


Figure 14.3. The UI of the *Smalltalk-80* IDE (1983) compared with the *Pharo* IDE (2017)

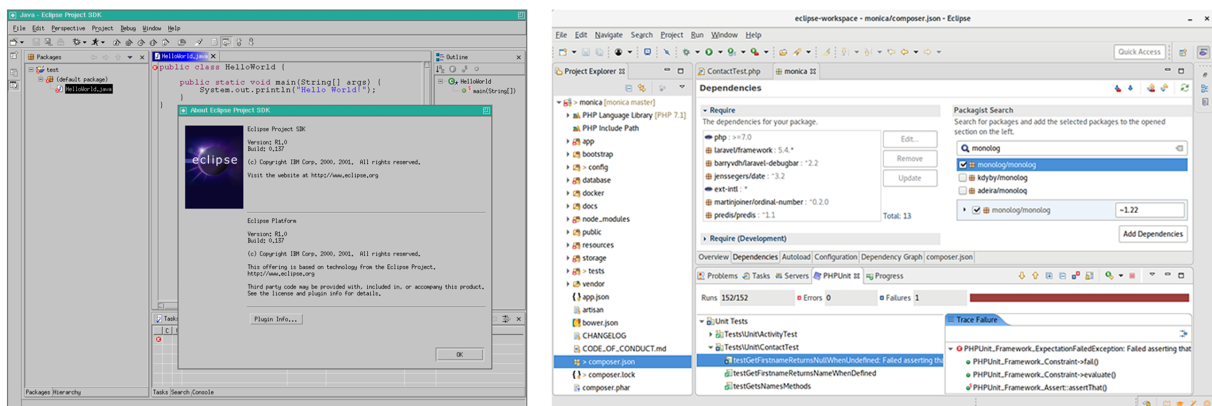


Figure 14.4. The UI of *Eclipse* 1.0 (2001) compared with *Eclipse* Oxygen (2017)

Novel UI Paradigms. Besides *Smalltalk-80* using grayscale and *Pharo* introducing a glimpse of colors, these two interfaces are largely the same. Researchers proposed alternative UIs to drift from traditional metaphors such as tab- and window-based interfaces. A notable example is CODE BUBBLES [BZR+10, BRZ+10]. As the name suggests, CODE BUBBLES displays code in bubbles instead of tabs or windows. Unfortunately, to the best of our knowledge, neither CODE BUBBLES nor other alternative UIs for IDEs have never taken over traditional IDEs.

Moldable Tools. More recently, Chiş *et al.* [Chi16] and Nierstrasz [Nie16] introduced *moldable* and context-aware tools as a solution to overcome the limitation of modern development environments. These tools are able to sense the underlying model and adapt the UI accordingly. For example, the *moldable debugger* is able to modify its behavior and changing its debugging operations according to the context [CNG13]. Chiş *et al.* provide examples on how their debugger behaves differently while debugging a parsing and a messaging framework inside the *Pharo* IDE.

Arranging UI Components. The UI components of the IDE, such as menus and buttons, are not always easily reachable. According to Lee *et al.*, for example, the support for refactoring offered by modern IDEs is inefficient [LCJ13]. Indeed, in the *Pharo* IDE the refactoring menu is not immediately visible. Interaction data can be used to detect when the developer is involved in refactoring activities or, in general, when she is struggling to find a particular menu item. An interaction-aware IDE could help developers reaching the UI components they are looking for or moving them to more convenient locations in the IDE.

Reducing the Entropy in the IDE. IDEs often force developers to open many windows (or tabs) to navigate to the entities of interest [RND09]. This quickly leads to an environment that is crowded with a high number of independent and apparently unrelated windows [DSE06]. Researcher called this phenomenon the *window plague* [RND09]. As a proof-of-concept, we developed a prototypical tool, called PLAGUE DOCTOR, that uses the fine-grained interactions to mitigate the *window plague* (see Chapter 12 for more information). In Chapter 13 we also discussed an approach to tame the UI of the IDE when the level of visual entropy is too high. According to our previous study, developers spend roughly 14% of their time moving around and resizing the UI components [MML15b]. We strongly believe that interaction data can be used to monitor, and control, the level of entropy in the IDE. The time spent in pure UI fiddling calls for novel interaction paradigms and UIs that minimize this waste of time.

Better Code Editors. Code editors (or system browsers in *Pharo* jargon) are among the most used UIs in the IDEs. Quoting Nierstrasz, however, they are "*basically glorified text editors*" [Nie16]. Our long term vision is to have code browsers that automatically reshape themselves to better support different activities, such as source code navigation [SES05]. For example, in Chapter 13 we described strategies to hide the parts of the windows that are not likely to be useful at any given moment to reduce the visual entropy of the IDE. Other basic improvements to existing code browsers concern, for example, how code entities are displayed. In *Pharo*, for example, browsers list entities in alphabetical order. We do not believe that this arrangement is optimal. With interaction data, for example, we can identify the working set [KM05, MML16b] and reorder these entities, according to the frequency of interactions (or changes).

14.5 Crowdsourced Holistic Mental Models

With an infrastructure such as EYE, described in Section 14.1, we can collect interactions of different developers with various information sources. With this information, we can develop the concept of *crowdsourced holistic mental models*, collaborative mental models that consider more than source code information. We can use this collaborative mental models, for example, to support developers in understanding hidden relationships between source code entities by augmenting the IDE with visualizations, color overlays, or recommendations.

Researchers proposed various approaches to enrich the UI of the IDE. AREAVIEW[4], for example, is a tool that visualizes metrics and "*areas of interest*" on top of UML architectural diagrams [BBT06, BT09]. Murphy-Hill and Black developed STENCH BLOSSOM [MHB10]. While the programmer is coding, this tool gives a visual high-level overview of the presence of *code smells* in the code base and provides means to understand their origin. We could take inspiration from these visual approaches and extend them to visualize other kinds of information obtained by leveraging crowdsourced holistic mental models.

14.6 Wrapping Up

Throughout our dissertation we stressed the fact that IDE interactions, largely neglected by modern IDEs, have a very high potential to both understand and support the workflow of developers. Our long-term vision includes different directions to further leverage this potential.

We envision an infrastructure to record the interactions with all the tools used during development, and not only IDE interactions. We discussed how interaction data can be used to create novel, and more effective, recommender systems and visualizations. Finally, we introduced the concept of crowdsourced holistic mental models and discussed how we can leverage IDE interactions to reshape the UI of the IDE.

[4]See `http://www.cs.rug.nl/~alext/SOFTWARE/ARCHIVIEW/`

15

Conclusions

We strongly believe that development environments should not neglect developer interactions but rather record and leverage them. Throughout this dissertation, we discussed the two main benefits that interaction-aware IDEs can bring to practitioners and developers. On the one side fine-grained IDE interaction data enable novel and in-depth analyses of the behavior of software developers. Likewise, interaction-aware IDEs can provide developers with effective and actionable support for their activities.

We set the ground for this dissertation by summarizing the history of software development, from punch cards to modern integrated development environments. Then we discussed the main source of information targeted by our research: the interactions between the developer and the *Pharo* IDE. We explained our reasons behind the choice of working in a not mainstream open-source development environment supported by an active community, and detailed its object model. We concluded the prologue of the dissertations by discussing privacy and ethics concerns arising from the collection and analyses of potential sensible data about software development and overviewing the state of the art in the fields connected to our research.

To validate our thesis, we developed DFlow, the main supporting tool of our research. DFlow silently collects development interactions in the *Pharo* IDE and makes them available for further use. On top of DFlow we developed approaches to support developers in dealing with the known limitations of modern development environments and used the data recorded with DFlow to study different aspects of the workflow of developers. Interaction data proved to be useful to both gain novel insights on software development and setting the ground for the next generation of tool support inside the IDE.

Structure of the Chapter

The chapter summarizes the contributions of this dissertation. It follows the same structure of the dissertation by presenting the contributions of each chapter grouping them in parts. Section 15.1 summarizes our contributions in recording and modeling interaction data. Section 15.2 presents the visual approaches we devised to better understand interaction data. In Section 15.3 we review our first steps towards leveraging interaction data in the IDE to support the development workflow. Finally, Section 15.4 recaps our vision for the future.

15.1 Modeling, Recording, and Interpreting Interaction Data

In Part II we discussed the foundations of our research: Recording, modeling, and interpreting interaction data inside the IDE. This is an essential step to make interaction data available for further use to support the workflow of developers.

15.1.1 DFlow: Our Interaction Profiler for the Pharo IDE

Chapter 5 introduced the main supporting tool for our research: DFlow. DFlow is the interaction profiler for the *Pharo* IDE that we developed. The chapter also described how DFlow and its architecture evolved over time. In its last version, DFlow automatically observes the interactions of the developer with the IDE, filters out the ones that are not interesting, and makes interesting development interactions available for further use. This architecture enables external tools to exploit the potential of the data captured by DFlow. The chapter also proposed a model to provide a unified structure to all interaction events recorded by DFlow. Our model identifies three different types of events: i) Meta, ii) User Input, and iii) User Interface Events. Meta events represent the interactions of developers with program entities, *e.g.,* classes and methods. User Input events are events performed using an input device, *e.g.,* mouse and keyboard. Finally, User Interface events are the interactions of the developer with the user interface of the IDE. For each category of events, the chapter listed the actual interaction events recorded by DFlow (see Table 5.1).

15.1.2 A Naïve Model to Interpret Interaction Data

In Chapter 6 we analyzed interaction data collected with two different IDE interaction profilers: DFlow and Plog, an interaction profiler for the *Eclipse* IDE [KKA12]. The chapter detailed an estimation model to assess how much time developers spend to navigate, write, and understand source code. In particular, our focus was to understand whether advances on software engineering practice changed the role of program comprehension with respect to what researchers claimed more than 30 years ago [ZSG79, FH83, Cor89]. Our findings suggested that the role of program comprehension has been significantly underestimated by previous research: On our dataset, program comprehension accounts on average from 54 to 94% of the total development time.

15.1.3 Inferring High-Level Development Activities from Interaction Histories

Motivated by the results of our naïve model (see Chapter 6), in Chapter 7 we developed a better estimation model to reconstruct high-level development activities from fine-grained IDE interaction histories. Since interactions happen instantaneously (*i.e.,* they have no duration), the goal of this model was to aggregate events into sequences (*i.e., sprees*) to precisely measure their duration. In practice, we devised rules to aggregate events into sprees, and later sprees into high-level development activities. Then, we decomposed development time into the following five categories: i) understanding, ii) navigation, iii) editing, iv) UI interactions, and v) time spent outside of the IDE. Among our results, we observed that our analysis attributes to program understanding more importance than what the common knowledge suggested, reaching a value of roughly 70%. Another interesting insight is that developers spent 17% of their time in fiddling with the user interface of the IDE, calling for novel IDE UI metaphors and support tools.

15.1.4 Measuring Navigation Efficiency in the IDE

Since navigation is an essential development activity [KMCA06], Chapter 8 focused on the use of interaction data to understand to what extent the *Pharo* IDE supports developers in navigating source code. We defined *navigation efficiency* as the ratio between an ideal navigation scenario and the actual behavior of the developer, encoded in the recorded interaction histories. The chapter explained the mechanics of navigation in the *Pharo* IDE and the ingredients we used to measure navigation efficiency: involved program entities and navigation cost. Then, we presented a naïve model, preliminary results, and discussed the limitations that led us to a more realistic model. Navigation efficiency depends on a variety of factors, and it is thus hard if not impossible to provide an exact estimation. However, our results showed that *Pharo* developers are by a large extent inefficient at navigating source code.

15.2 Visual Analytics of Development Sessions

Part III of our dissertation discussed various visual approaches to gather further insights from interaction histories using software visualizations.

15.2.1 Understanding How Developers Use the User Interface of the IDE

In Chapter 9 we presented a visual approach to better understand how developers use the user interface of the *Pharo* IDE. Our approach is based on a visualization composed of two parts: UI View and an Activity Timeline. The former shows how developers interact with different UIs, and how the work is essentially organized in development tracks. The Activity Timeline, instead, depicts development activities and enabling to understand how they relate to UI usage. We used the visualization to tell four development stories that highlighted both peculiar developer behaviors on the usage of the IDE and their activities, and well known phenomena like the *window plague* [RND09]. The chapter also proposed a visual classification of development sessions in terms of dominant window tracks and flow between these tracks. We identified "conservative" developers using a small number of windows and "frenetic" developers who continuously open new windows for each task.

15.2.2 Visualizing the Evolution of Working Sets

Chapter 10 presented a visual approach to understand the evolution of the working set during development sessions. We called *working set* a group of program entities which a developer has interacted with during a particular period of time. Our visualization distinguished between current and past working sets and used two different layouts to arrange the two disjoints groups of entities. We applied our visualization to 914 development sessions coming from 14 developers and discused a catalogue of visual patterns on the evolution of working sets during development.

15.2.3 Other Visualizations and Storytelling

After we developed the initial prototype of DFlow and collected interaction data from the first handful of developers, we discovered that raw interaction data are very hard to interpret. To get an initial understanding of the data, we devised a basic catalog of software visualizations to depict interaction histories from different perspective, described in Chapter 11. The chapter also detailed DFloWeb, an early experiment on visualizing the workflow of developers in the web.

We used the visualizations to identify development stories and to gather interesting insights from the recorded development sessions.

15.3 Supporting Developers with Interaction Data

In Part IV of the dissertation we described our first steps in leveraging interaction data to support software development inside the IDE.

15.3.1 The Plague Doctor: Curing the Window Plague

Developers are often confronted with a large number of windows (or tabs) inside the IDE, most of which are irrelevant for the current development session [RND09]. This phenomenon is known as the *window plague* [RND09]. In Chapter 12 we presented the PLAGUE DOCTOR, a tool that aims to mitigate this plague by decorating the windows inside the IDE and providing developers with suggestions on which windows are potentially irrelevant for the current development task. Our tool is inspired by AUTUMN LEAVES, developed by Röthlisberger *et al.* [RND09], but it differs from it in several ways, *e.g.,* the better quality of interaction data it leverages and the customization possibility it offers (see Section 12.1.2, for more information).

15.3.2 Taming the User Interface of the IDE

Due to several factors, including the intrinsic complexity of software development, the user interface of IDEs become very chaotic, potentially hindering the productivity of developers. Chapter 13 provided evidence of chaos by analyzing data coming from two different IDEs: *Pharo* and *Eclipse*. *Pharo* interaction histories were recorded with DFLOW, while for *Eclipse* we used the publicly available MYLYN dataset obtained from the *Eclipse* Bugzilla repository. In addition, by using the more fine-grained interaction histories of DFLOW, we were able to characterize the level of chaos in terms of the window space required to support development tasks and the amount of overlapping of these windows. Our results showed that, on average, developers inherently spent 30% of their time in a chaotic environment. To this aim, the chapter also proposed an approach to reduce the cluttering of the *Pharo* IDE potentially making it a more pleasant working environment.

15.4 Our Vision for the Future

Chapter 14 concluded our dissertation by outlining possible research directions for IDEs and tools of the future. In particular, we proposed a holistic profiler infrastructure that observes and leverages the interactions of the developer with all the applications she uses (*e.g.,* IDE, web browser, mail client). Our long-term vision included more actionable visualizations inside the IDE to support a variety of activities and novel solutions for the—currently outdated—user interfaces of modern IDEs. We concluded the chapter by discussing the concept of "*crowdsourced holistic mental model*". In this vision, the construction of mental models becomes a collaborative activity involving different developers and various information sources besides the overused—and old fashioned—source code change data.

15.5 Closing Words

In this dissertation, we discussed how interaction-aware IDEs can be beneficial for both practitioners and developers. Collecting IDE interactions enables novel and in-depth analyses of the behavior of software developers. Moreover, IDEs that are aware of those interactions can provide developers with effective and actionable support for their development activities.

Our contributions can be grouped into three areas: i) approaches to model, record, and interpret interaction data; ii) visual analytics of interaction histories; and iii) techniques to leverage IDE interactions to support the development workflow. According to Snipes *et al.*, "*the next two decades will surely be labeled as the era of smarter big data analysis*" [SMHF+15]. With our work we set foot in this *era*. Now, we look forward to the next decades to see the rise of novel approaches to exploit the full potential of interaction data.

Bibliography

[ABSN13] Chee Siang Ang, Ania Bobrowicz, Diane J. Schiano, and Bonnie Nardi. Data in the Wild: Some Reflections. *ACM Interactions*, 20(2):39–43, March 2013.

[AFQ+15] Vinay Augustine, Patrick Francis, Xiao Qu, David Shepherd, Will Snipes, Christoph Bräunlich, and Thomas Fritz. A Field Study on Fostering Structural Navigation with Prodet. In *Proceedings of ICSE 2015 (37th International Conference on Software Engineering)*, pages 229–238. IEEE, 2015.

[APNM16] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A Study of Visual Studio Usage in Practice. In *Proceedings of SANER 2016 (23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, pages 124–134. IEEE, 2016.

[BAB+93] David M. Butler, James C. Almond, R. Daniel Bergeron, Ken W. Brodlie, and Robert B. Haber. Visualization Reference Models. In *Proceedings of VIS 1993 (4th Conference on Visualization)*, pages 337–342. IEEE CS Press, 1993.

[Bac78] John Backus. The History of FORTRAN I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, August 1978.

[BBRR12] Alexandre Bergel, Felipe Banados, Romain Robbes, and David Röthlisberger. Spy: A Flexible Code Profiling Framework. *Computer Languages, Systems & Structures*, 38(1):16–28, 2012.

[BBT06] Heorhiy Byelas, Egor Bondarev, and Alexandru Telea. Visualization of Areas of Interest in Component-Based System Architectures. In *Proceedings of SEAA 2006 (32nd EUROMICRO Conference on Software Engineering and Advanced Applications)*, pages 160–169. IEEE, 2006.

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile Visualization with Roassal. In *Deep Into Pharo*, chapter 11, pages 209–239. Square Bracket Associates, 2013.

[Ben12] Peter J. Bentley. *Digitized: The Science of Computers and How It Shapes Our World*. Oxford University Press, 2012.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Donald Roberts. Wrappers to the Rescue. In *Proceedings of ECOOP 1998 (12th European Conference on Object-Oriented Programming)*, pages 396–417. Springer, 1998.

[BGL+09] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of CHI 2009 (27th Conference on Human Factors in Computing Systems)*, pages 1589–1598. ACM, 2009.

[BGMW16] Bo Brinkman, Don Gotterbarn, Keith Miller, and Marty J. Wolf. Making a Positive Impact: Updating the ACM Code of Ethics. *Communications of the ACM*, 59(12):7–13, December 2016.

[BHvW99] Mark Bruls, Kees Huizing, and Jarke van Wijk. Squarified Treemaps. In *Proceedings of VisSym 1999 (1st Joint Eurographics and IEEE TCVG Symposium on Visualization)*, pages 33–42. Eurographics Association, 1999.

[BK06] Brian P. Bailey and Joseph A. Konstan. On the Need for Attention-Aware Systems: Measuring Effects of Interruption on Task Performance, Error Rate, and Affective State. *Computers in Human Behavior*, 22(4):685–708, 2006.

[Bro85] Frederick P. Brooks. *The Mythical Man-month.* Addison-Wesley, 1985.

[Bro95] Frederick P. Brooks. *The Mythical Man-Month.* Addison-Wesley, 2nd edition, 1995.

[BRZ$^+$10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering)*, pages 455–464. ACM/IEEE, 2010.

[BT09] Heorhiy Byelas and Alexandru Telea. Visualizing Metrics on Areas of Interest in Software Architecture Diagrams. In *Proceedings of PacificVis 2009 (IEEE Pacific Visualization Symposium)*, pages 33–40. IEEE, 2009.

[BZR$^+$10] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. In *Proceedings of CHI 2010 (28th International Conference on Human Factors in Computing Systems)*, pages 2503–2512. ACM, 2010.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[CG07] Maurice M. Carey and Gerald C. Gannod. Recovering Concepts from Source Code with Automated Concept Identification. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 27–36. IEEE, 2007.

[Chi16] Andrei Chiş. *Moldable Tools.* PhD thesis, University of Bern, 2016.

[Chr08] Clifford G. Christians. Ethics and Politics in Qualitative Research. In Denzin, Norman K. and Lincoln, Yvonna S., editor, *The Landscape of Qualitative Research*, volume 1. Sage, 2008.

[Cir09] Francesco Cirillo. *The Pomodoro Technique.* FC Garage GmbH, 2009.

[CNG13] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. Towards a Moldable Debugger. In *Proceedings of DYLA 2013 (7th Workshop on Dynamic Languages and Applications)*, pages 2:1–2:6. ACM, 2013.

[Cor89] Thomas A. Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[CRC07] Alan Cooper, Robert Reimann, and David Cronin. *About Face 3: The Essentials of Interaction Design.* John Wiley & Sons, 2007.

[CS08] Irina D. Coman and Alberto Sillitti. Automated Identification of Tasks in Development Sessions. In *Proceedings of ICPC 2008 (16th International Conference on Program Comprehension)*, pages 212–217. IEEE, 2008.

[Csi90] Mihaly Csikszentmihalyi. *Flow - The Psychology of Optimal Experience.* Harper Perennial, 1990.

[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 ICSE 2012 (34th International Conference on Software Engineering)*, pages 1064–1073. IEEE, 2012.

[DCR05] Robert DeLine, Mary Czerwinski, and George G Robertson. Easing Program Comprehension by Sharing Navigation Data. In *Proceedings of VL/HCC 2005 (Symposium on Visual Languages and Human-Centric Computing)*, pages 241–248. IEEE, 2005.

[DCSP16] Kostadin Damevski, Hui Chen, David Shepherd, and Lori Pollock. Interactive Exploration of Developer Interaction Traces Using a Hidden Markov Model. In *Proceedings of MSR 2016 (13th IEEE Working Conference on Mining Software Repositories)*, pages 126–136. ACM, 2016.

[Den08] Marc Denker. *Sub-method Structural and Behavioral Reflection.* PhD thesis, University of Bern, 2008.

[DR10] Robert DeLine and Kael Rowan. Code Canvas: Zooming Towards Better Development Environments. In *Proceedings of ICSE 2010 (32nd International Conference on Software Engineering) – ERA*, pages 207–210. ACM/IEEE, 2010.

[DRW00] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE 2000 (*22nd *International Conference on Software Engineering)*, pages 467–476. ACM, 2000.

[DSE06] Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid Source Code Views. In *Proceedings of ICPC 2006 (14th IEEE International Conference on Program Comprehension)*, pages 260–263. IEEE, 2006.

[DSML15] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. Misery Loves Company - CrowdStacking Traces to Aid Problem Detection. In *Proceedings of SANER 2015 (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, pages 131–140. IEEE, 2015.

[DSSP17] Kostadin Damevski, David C. Shepherd, Johannes Schneider, and Lori Pollock. Mining Sequences of Developer Interactions in Visual Studio for Usage Smells. *IEEE TSE 2017 (Transactions on Software Engineering)*, 43(4):359–371, April 2017.

[DZHC17] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. *Pharo by Example 5.0.* Square Bracket Associates, 2017.

[EHRS14] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do API Documentation and Static Typing Affect API Usability? In *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*, pages 632–642. ACM, 2014.

[Erl00] Len Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23, May 2000.

[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 ICSE 2014 (36th International Conference on Software Engineering)*, pages 402–413. ACM/IEEE, 2014.

[FF03] John Fuegi and Jo Francis. Lovelace & Babbage and the Creation of the 1843 'Notes'. *IEEE Annals of the History of Computing*, 25(4):16–26, Oct 2003.

[FGS11] Tim Frey, Marius Gelhausen, and Gunter Saake. Categorization of Concerns: A Categorical Program Comprehension Model. In *Proceedings of PLATEAU 2011 (3rd Workshop on Evaluation and Usability of Programming Languages and Tools)*, pages 73–82. ACM, 2011.

[FH83] Richard K. Fjeldstad and William T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In Girish Parikh and Nicholas Zvegintzov, editor, *Tutorial on Software Maintenance*, pages 13–30. IEEE, 1983.

[Fit54] Paul M Fitts. The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement. *Journal of Experimental Psychology*, 47(6):381, 1954.

[FM16] Thomas Fritz and Sebastian C. Müller. Leveraging Biometric Data to Boost Software Developer Productivity. In *Proceedings of SANER 2016 (23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, pages 66–77. IEEE, 2016.

[FSK+14] Thomas Fritz, David C. Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich. Developers' Code Context Models for Change Tasks. In *Proceedings of FSE (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 7–18. ACM, 2014.

[Gee05] David D. Geer. Eclipse Becomes the Dominant Java IDE. *IEEE Computer*, 38(7):16–18, July 2005.

[GGD07] Orla Greevy, Tudor Gîrba, and Stéphane Ducasse. How Developers Develop Features. In *Proceedings of CSMR 2007 (11th European Conference on Software Maintenance and Reengineering)*, pages 265–274. IEEE, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How Developers Drive Software Evolution. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 113–122. IEEE CS Press, 2005.

[GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the Evolution of Class Hierarchies. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering)*, pages 2–11. IEEE CS Press, 2005.

[Gra08] Paul Graham. *Hackers & painters: Big Ideas from the Computer Age*. O'Reilly Media, Inc., 2008.

[GSBS14] Zhongxian Gu, Drew Schleck, Earl T. Barr, and Zhendong Su. Capturing and Exploiting IDE Interactions. In *Proceedings of Onward! 2014 (ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software)*, pages 83–94. ACM Press, 2014.

[HC86] D. Austin Henderson and Stuart Card. Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface. *ACM Transactions on Graphics*, 5(3):211–243, 1986.

[HHG46] Adele Goldstine Herman H. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, 1946.

[HKR+14] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[HS95] Brian Henderson-Sellers. *Object-oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., 1995.

[Hum02] Watts S. Humphrey. Personal Software Process (PSP). In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[INM+05] Florian Iragne, Macha Nikolski, Bertrand Mathieu, David Auber, and David Sherman. ProViz: Protein Interaction Visualization and Exploration. *Bioinformatics*, 21(2):272–274, 2005.

[JD03] Doug Janzen and Kris De Volder. Navigating and Querying Code without Getting Lost. In *Proceedings of AOSD 2003 (2nd International Conference on Aspect-Oriented Software Development)*, pages 178–187. ACM, 2003.

[JKA+03] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William E.J. Doane. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In *Proceedings of ICSE 2003 (25th International Conference on Software Engineering)*, pages 641–646. IEEE CS Press, 2003.

[JKA+04] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Practical Automated Process and Product Metric Collection and Analysis in a Classroom Setting: Lessons Learned from Hackystat-UH. In *Proceedings of ISESE 2004 (International Symposium on Empirical Software Engineering)*, pages 136–144. IEEE, 2004.

[Joh99] Deborah G. Johnson. Sorting Out the Uniqueness of Computer-Ethical Issues. *Etica & Politica*, 1(2), 1999.

[Joh01] Philip M. Johnson. You Can't Even Ask Them to Push a Button: Toward Ubiquitous, Developer-Centric, Empirical Software Engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*. NCO NITRD, 2001.

[Jon78] Thomas C. Jones. Measuring Programming Quality and Productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[Kay93] Alan C. Kay. The Early History of Smalltalk. *ACM SIGPLAN Notices*, 28(3):69–95, March 1993.

[KBC+15] Juraj Kubelka, Alexandre Bergel, Andrei Chiş, Tudor Gîrba, Stefan Reichhart, Romain Robbes, and Aliaksei Syrel. On Understanding How Developers Use the Spotter Search Tool. In *Proceedings of VISSOFT 2015 (3rd Working Conference on Software Visualization)*, pages 145–149. IEEE, 2015.

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of ICSE 2007 (29th International Conference on Software Engineering)*, pages 344–353. IEEE CS Press, 2007.

[KKA12] Takashi Kobayashi, Nozomu Kato, and Kiyoshi Agusa. Interaction Histories Mining for Software Change Guide. In *Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, pages 73–77. IEEE, 2012.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP 1999 (11th European Conference on Object-Oriented Programming)*, pages 220–242. Springer, 1997.

[KM05] Mik Kersten and Gail C. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of AOSD 2005 (4th International Conference on Aspect-Oriented Software Development*, pages 159–168. ACM, 2005.

[KM06] Mik Kersten and Gail C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of FSE 2006 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 1–11. ACM, 2006.

[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 TSE 2006 (Transactions on Software Engineering)*, 32(12):971–987, 2006.

[Koh93] Alfie Kohn. *Punished by Rewards*. Houghton Mifflin, 1993.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[Lan05] Marc Langheinrich. Privacy Mechanisms and Principles. In *Personal Privacy in Ubiquitous Computing Tools and System Support (PhD Thesis)*, chapter 3, pages 45–114. ETH Zurich, 2005.

[LB12] Howard Lune and Bruce Lawrence Berg. Ethical Issues. In *Qualitative Research Methods for the Social Sciences*, chapter 3, pages 61–104. Pearson Education, 12 edition, 2012.

[LCJ13] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering)*, pages 23–32. IEEE, 2013.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE TSE 2003 (Transactions on Software Engineering)*, 29(9):782–795, September 2003.

[Les99] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, 1999.

[LHB03] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design*. Rockport, 2003.

[LVD06] Thomas LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, pages 492–501. ACM/IEEE, 2006.

[Mac67] James MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the* 5th *Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[Mas14] Ebrisa Mastrodicasa. Ludus Opus Proficit - A Gamification Framework for Software Engineering. Master's thesis, Università della Svizzera italiana (USI), 2014.

[MBML14] Roberto Minelli, Lorenzo Baracchi, Andrea Mocci, and Michele Lanza. Visual Storytelling of Development Sessions. In *Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution, ERA Track)*, pages 416–420. IEEE, 2014.

[McC99] Scott McCartney. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Walker & Company, 1999.

[McG11] Jane McGonigal. *Reality is Broken*. Penguin, 2011.

[MFR14] Walid Maalej, Thomas Fritz, and Romain Robbes. Collecting and Processing Interaction Data for Recommendation Systems. In Martin P. Robillard and Walid Maalej and Robert J. Walker and Thomas Zimmermann, editor, *Recommendation Systems in Software Engineering*, chapter 7, pages 173–197. Springer, 2014.

[MGH05] Gloria Mark, Victor M Gonzalez, and Justin Harris. No Task Left Behind? Examining the Nature of Fragmented Work. In *Proceedings of SIGCHI 2005 (Conference on Human Factors in Computing Systems)*, pages 321–330. ACM, 2005.

[MHB10] Emerson Murphy-Hill and Andrew P. Black. An Interactive Ambient Visualization for Code Smells. In *Proceedings of SOFTVIS 2010 (5th International Symposium on Software Visualization)*, pages 5–14. ACM, 2010.

[Min14] Roberto Minelli. Towards Self-Adaptive IDEs. In *Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution, Doctoral Symposium)*, page 666. IEEE, 2014.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[MKRČ05] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubranić. The Emergent Structure of Development Tasks. In *Proceedings of ECOOP 2005 (19th European Conference on Object-Oriented Programming)*, pages 33–48. Springer, 2005.

[ML13a] Roberto Minelli and Michele Lanza. DFlow - Towards the Understanding of the Workflow of Developers. In *Proceedings of SATTOSE 2013 (6th Seminar Series on Advanced Techniques & Tools for Software Evolution)*, 2013.

[ML13b] Roberto Minelli and Michele Lanza. Visualizing the Workflow of Developers. In *Proceedings of VISSOFT 2013 (1st IEEE Working Conference on Software Visualization)*, pages 1–4. IEEE, 2013.

[MML15a] Roberto Minelli, Andrea Mocci, and Michele Lanza. Free Hugs – Praising Developers For Their Actions. In *Proceedings of ICSE 2015 (37th International Conference on Software Engineering)*, pages 555–558. IEEE, 2015.

[MML15b] Roberto Minelli, Andrea Mocci, and Michele Lanza. I Know What You Did Last Summer – An Investigation of How Developers Spend Their Time. In *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*, pages 25–35. IEEE Press, 2015.

[MML15c] Roberto Minelli, Andrea Mocci, and Michele Lanza. The Plague Doctor: A Promising Cure for the Window Plague. In *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*, pages 182–185. IEEE Press, 2015.

[MML16a] Roberto Minelli, Andrea Mocci, and Michele Lanza. Measuring Navigation Efficiency in the IDE. In *Proceedings of IWESEP 2016 (7th IEEE International Workshop on Empirical Software Engineering in Practice)*, pages 1–6. IEEE, 2016.

[MML16b] Roberto Minelli, Andrea Mocci, and Michele Lanza. Visualizing the Evolution of Working Sets. In *Proceedings of VISSOFT 2016 (4th IEEE Working Conference on Software Visualization)*, pages 141–150. IEEE, 2016.

[MMLB14] Roberto Minelli, Andrea Mocci, Michele Lanza, and Lorenzo Baracchi. Visualizing Developer Interactions. In *Proceedings of VISSOFT 2014 (2nd IEEE Working Conference on Software Visualization)*, pages 147–156. IEEE, 2014.

[MMLK14] Roberto Minelli, Andrea Mocci, Michele Lanza, and Takashi Kobayashi. Quantifying Program Comprehension with Interaction Data. In *Proceedings of QSIC 2014 (14th International Conference on Quality Software)*, pages 276–285. IEEE, 2014.

[MMRL16] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. Taming the IDE with Fine-grained Interaction Data. In *Proceedings of ICPC 2016 (24th IEEE International Conference on Program Comprehension)*, pages 1–10. IEEE, 2016.

[Moo01] Leon Moonen. Generating Robust Parsers Using Island Grammars. In *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*, pages 13–22. IEEE, 2001.

[NCDJ14] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*, pages 803–813. ACM, 2014.

[Nie16] Oscar Nierstrasz. The Death of Object-Oriented Programming. In *Proceedings of FASE 2016 (19th International Conference on Fundamental Approaches to Software Engineering)*, pages 3–10. Springer, 2016.

[OLDR11] Fernando Olivero, Michele Lanza, Marco D'Ambros, and Romain Robbes. Enabling Program Comprehension through a Visual Object-Focused Development Environment. In *Proceedings of VL/HCC 2011 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, pages 127–134. IEEE, 2011.

[OM09] Michael Ogawa and Kwan-Liu Ma. code_swarm: A Design Study in Organic Software Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1097–1104, Nov 2009.

[OM10] Michael Ogawa and Kwan-Liu Ma. Software Evolution Storylines. In *Proceedings of SOFTVIS 2010 (5th International Symposium on Software Visualization)*, pages 35–42. ACM, 2010.

[PBL13] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. SeaHawk: Stack Overflow in the IDE. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering, Tool Demo Track)*, pages 1295–1298. IEEE CS Press, 2013.

[PBS93] Blaine Price, Ronald Baecker, and Ian Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, pages 211–266, 1993.

[PFK+13] David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. The Whats and Hows of Programmers' Foraging Diets. In *Proceedings of SIGCHI 2013 (Conference on Human Factors in Computing Systems)*, pages 3063–3072. ACM, 2013.

[PFS+11] David J. Piorkowski, Scott D. Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E. John, Margaret M. Burnett, and Rachel Bellamy. Modeling Programmer Navigation: A Head-To-Head Empirical Evaluation of Predictive Models. In *Proceedings of VL/HCC 2011 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, pages 109–116. IEEE, 2011.

[PG06] Chris Parnin and Carsten Görg. Building Usage Contexts During Program Comprehension. In *Proceedings of ICPC 2006 (14th IEEE International Conference on Program Comprehension)*, pages 13–22. IEEE, 2006.

[PHR14] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. In *Proceedings of ICPC (22nd International Conference on Program Comprehension)*, pages 212–222. ACM, 2014.

[Pin99] Steven Pinker. How the Mind Works. *Annals of the New York Academy of Sciences*, 882(1):119–127, 1999.

[PSB+17] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, and Michele Lanza. Supporting Software Developers with a Holistic Recommender System. In *Proceedings of ICSE 2017 (39th International Conference on Software Engineering)*, pages 94–105. IEEE Press, 2017.

[PY07] Mauro Pezzè and Michal Young. Structural Testing. In *Software Testing and Analysis: Process, Principles and Techniques*, chapter 12. Wiley, 2007.

[RCM04] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE TSE 2004 (Transactions on Software Engineering)*, 30(12):889–903, 2004.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[RL07] Romain Robbes and Michele Lanza. Characterizing and Understanding Development Sessions. In *Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension)*, pages 155–164. IEEE CS Press, 2007.

[RL08] Romain Robbes and Michele Lanza. SpyWare: A Change-Aware Development Toolset. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*, pages 847–850. ACM Press, 2008.

[RL10] Romain Robbes and Michele Lanza. How Program History Can Improve Code Completion. *Journal of Automated Software Engineering*, 17(2):181–212, 2010.

[RM03] Martin P. Robillard and Gail C. Murphy. Automatically Inferring Concern Code from Program Investigation Activities. In *Proceedings of ASE 2003 (18th IEEE International Conference on Automated Software Engineering)*, pages 225–234. IEEE, 2003.

[RM07] Martin P. Robillard and Gail C. Murphy. Representing Concerns in Source Code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):3, 2007.

[RND09] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. Autumn Leaves: Curing the Window Plague in IDEs. In *Proceedings of WCRE 2009 (16th Working Conference on Reverse Engineering)*, pages 237–246. IEEE, 2009.

[RND11] David Röthlisberger, Oscar Nierstrasz, and Stephane Ducasse. SmartGroups: Focusing on Task-Relevant Source Artifacts in IDEs. In *Proceedings of ICPC 2011 (19th International Conference on Program Comprehension)*, pages 61–70. IEEE, 2011.

[RPL10] Romain Robbes, Damien Pollet, and Michele Lanza. Replaying IDE Interactions to Evaluate and Improve Change Prediction Approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 161–170. IEEE CS Press, 2010.

[RvDR+00] George Robertson, Maarten van Dantzich, Daniel Robbins, Mary Czerwinski, Ken Hinckley, Kirsten Risden, David Thiel, and Vadim Gorokhovsky. The Task Gallery: A 3D Window Manager. *Proceedings of SIGCHI 2000 (ACM Conference on Human Factors in Computing Systems*, 2(1):494–501, 2000.

[RWZ10] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, 2010.

[Sam69] Jean E. Sammet. *Programming Languages: History and Fundamentals.* Prentice-Hall, Inc., 1969.

[SBM01] Margaret-Anne Storey, Casey Best, and Jeff Michand. SHriMP Views: An Interactive Environment for Exploring Java Programs. In *Proceedings of IWPC 2001 (9th International Workshop on Program Comprehension)*, pages 1–4. IEEE, 2001.

[SCG$^+$15] Aliaksei Syrel, Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, and Stefan Reichhart. Spotter: Towards a Unified Search Interface in IDEs. In *Companion Proceedings of SPLASH 2015 (ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity)*, pages 54–55. ACM, 2015.

[SDBE98] John Stasko, John Domingue, Marc Brown, and Blaine Price (Eds.). *Software Visualization - Programming as a Multimedia Experience.* The MIT Press, 1998.

[Seb12] Robert W. Sebesta. *Concepts of Programming Languages.* Pearson, 10 edition, 2012.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting Navigation in Software Maintenance. In *Proceedings of ICSM 2005 (21st International Conference on Software Maintenance)*, pages 325–334. IEEE, 2005.

[SFM99] Margaret-Anne Storey, F. D. Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[Sha07] Yuval Sharon. EclipsEye - Spying on Eclipse. Bachelor's Thesis, Università della Svizzera italiana (USI), June 2007.

[SHFL16] Alka Singh, Austin Henley, Scott Fleming, and Maria Luong. An Empirical Evaluation of Models of Programmer Navigation. In *Proceedings of ICSME 2016 (International Conference on Software Maintenance and Evolution)*, pages 9–19. IEEE, 2016.

[SJ13] Francisco Servant and James A. Jones. Chronos: Visualizing Slices of Source-Code History. In *Proceedings of VISSOFT 2013 (1st IEEE Working Conference on Software Visualization)*, pages 1–4. IEEE, 2013.

[SJSV03] Alberto Sillitti, Andrea Janes, Giancarlo Succi, and Tullio Vernazza. Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. In *Proceedings of EUROMICRO 2003 (29th EUROMICRO Conference)*, pages 336–342. IEEE, 2003.

[SKG$^+$13] Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Bram Adams. On the Effect of Program Exploration on Maintenance Tasks. In *Proceedings of WCRE 2013 (20th Working Conference on Reverse Engineering)*, pages 391–400. IEEE, 2013.

[Ski78] Burrhus Skinner. *Reflections on Behaviorism and Society.* Prentice Hall, 1978.

[SL14] Tommaso Dal Sasso and Michele Lanza. in*Bug: Visual Analytics of Bug Repositories. In *Proceedings of CSMR-WCRE 2014 (1st Joint Meeting of the European Conference on Software Maintenance and Reengineering and the Working Conference on Reverse Engineering)*, pages 415–419. IEEE, 2014.

[SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An Examination of Software Engineering Work Practices. In *Proceedings of CASCON 1997 (Conference of the Centre for Advanced Studies on Collaborative Research)*, pages 21–36. IBM Corp., 1997.

[SM95] Margaret-Anne Storey and Hausi A. Müller. Manipulating and Documenting Software Structures Using SHriMP Views. In *Proceedings of ICSM 1995 (International Conference on Software Maintenance)*, pages 275–284. IEEE, 1995.

[SMDV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and Answering Questions During a Programming Change Task. *IEEE TSE 2008 (Transactions on Software Engineering)*, 34(4):434–451, 2008.

[SMHF$^+$15] Will Snipes, Emerson Murphy-Hill, Thomas Fritz, Mohsen Vakilian, Kostadin Damevski, Anil R. Nair, and David Shepherd. A Practical Guide to Analyzing IDE Usage Data. In Christian Bird and Tim Menzies and Thomas Zimmermann, editor, *The Art and Science of Analyzing Software Data*, chapter 5, pages 85–138. Elsevier, 2015.

[SMML15] Tommaso Dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. Blended, Not Stirred: Multi-Concern Visualization of Large Software Systems. In *Proceedings of VISSOFT 2015 (3rd IEEE Working Conference on Software Visualization)*, pages 106–115. IEEE, 2015.

[SNMH14] Will Snipes, Anil R. Nair, and Emerson Murphy-Hill. Experiences Gamifying Developer Adoption of Practices and Tools. In *Companion Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*, pages 105–114. ACM, 2014.

[SRG15] Heider Sanchez, Romain Robbes, and Victor M. Gonzalez. An Empirical Study of Work Fragmentation in Software Evolution Tasks. In *Proceedings of SANER 2015 (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, pages 251–260. IEEE, 2015.

[SSA15] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of ICSE 2015 (37th IEEE/ACM International Conference on Software Engineering)*, pages 9–19. IEEE, 2015.

[Sui05] Bernard Suits. *The Grasshopper: Games, Life and Utopia*. Broadview Press, 2005.

[TA08] Alexandru Telea and David Auber. Code Flows: Visualizing Structural Evolution of Source Code. In *Proceedings of EuroVis 2008 (10th Joint Eurographics / IEEE - VGTC Conference on Visualization)*, pages 831–838. Eurographics Association, 2008.

[Tay94] Ralph B. Taylor. *Research Methods in Criminal Justice*. McGraw-Hill, 1994.

[TFMH10] Gail C. Murphy Thomas Fritz, Jingwen Ou and Emerson Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 385–394. IEEE CS Press, 2010.

[TK03] Richard Thomas and Gregor Kennedy. Generic Usage Monitoring of Programming Students. In *Proceedings of ASCILITE 2003 (23th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education)*, pages 7–10. ASCILITE, 2003.

[TMN+99] Koji Torii, Ken-ichi Matsumoto, Kumiyo Nakakoji, Yoshihiro Takada, Shingo Takada, and Kazuyuki Shima. Ginger2: An Environment for Computer-Aided Empirical Software Engineering. *IEEE TSE 1999 (Transactions on Software Engineering)*, 25(4):474–492, 1999.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proceedings of OOPSLA 2003 (18th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 27–46. ACM, 2003.

[Tri06] Mario Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[TS02] John M. T. Thompson and H. Bruce Stewart. *Nonlinear Dynamics and Chaos*. John Wiley & Sons, 2002.

[VCN+11] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Roshanak Zilouchian Moghaddam, and Ralph E. Johnson. The Need for Richer Refactoring Usage Data. In *Proceedings of PLATEAU 2011 (3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools)*, pages 31–38. ACM, 2011.

[VFS13] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge. In *Proceedings of SocialCom 2013 (International Conference on Social Computing)*, pages 188–195. IEEE, 2013.

[VMV95] Anneliese Von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, Aug 1995.

[vN93] John von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, October 1993.

[Wal03] Andrew Walenstein. Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering. In *Proceedings of IWPC 2003 (11th International Workshop on Program Comprehension)*, pages 185–194. IEEE, 2003.

[Wei85] Gerald M. Weinberg. *The Psychology of Computer Programming*. John Wiley & Sons, Inc., 1985.

[WH92] Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE TSE 1992 (Transactions on Software Engineering)*, 18(12):1038–1044, 1992.

[WH12] Kevin Werbach and Dan Hunter. *For the Win*. Wharton Digital Press, 2012.

[WL07] Richard Wettel and Michele Lanza. Program Comprehension through Software Habitability. In *Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension)*, pages 231–240. IEEE, 2007.

[WM99] Alan Wexelblat and Pattie Maes. Footprints: History-Rich Tools for Information Foraging. In *Proceedings of CHI 1999 (International Conference on Human Factors in Computing Systems)*, pages 270–277. ACM, 1999.

[Wol16] Marty J. Wolf. The ACM Code of Ethics: A Call to Action. *Communications of the ACM*, 59(12):6–6, December 2016.

[YM11] YoungSeok Yoon and Brad A. Myers. Capturing and Analyzing Low-Level Events from the Code Editor. In *Proceedings of PLATEAU 2011 (3rd Workshop on Evaluation and Usability of Programming Languages and Tools)*, pages 25–30. ACM, 2011.

[YMK13] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. Visualization of Fine-Grained Code Change History. In *Proceedings of VL/HCC 2013 (IEEE Symposium on Visual Languages and Human-Centric Computing)*, pages 119–126. IEEE, 2013.

[YR11] Annie T. T. Ying and Martin P. Robillard. The Influence of the Task on Programmer Behaviour. In *Proceedings of ICPC 2011 (19th International Conference on Program Comprehension)*, pages 31–40. IEEE, 2011.

[ZCM^{+}17] Manuela Züger, Christopher Corley, Andre N Meyer, Boyang Li, Thomas Fritz, David Shepherd, Vinay Augustine, Patrick Francis, Nicholas Kraft, and Will Snipes. Reducing Interruptions at Work: A Large-Scale Field Study of FlowLight. In *Proceedings of CHI 2017 (Conference on Human Factors in Computing Systems)*, pages 61–72. ACM, 2017.

[ZF15] Manuela Züger and Thomas Fritz. Interruptibility of Software Developers and Its Prediction Using Psycho-Physiological Sensors. In *Proceedings of CHI 2015 (33rd Annual ACM Conference on Human Factors in Computing Systems)*, pages 2981–2990. ACM, 2015.

[ZG06] Lijie Zou and Michael W. Godfrey. An Industrial Case Study of Program Artifacts Viewed During Maintenance Tasks. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 71–82. IEEE, 2006.

[ZSG79] Marvin Zelkowitz, Alan Shaw, and John Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.

Online Resources

[ABB17] ABB Inc. ABB Dev Interaction Data – Over 30,000 Hours of Developer Interaction Data in Visual Studio Captured with the Blaze Tool. `https://abb-iss.github.io/DeveloperInteractionLogs`, 2017. Accessed: 2017-05-03.

[Car16] Pierre Carbonnelle. Top IDE Index. `https://pypl.github.io/IDE.html`, 2016. Accessed: 2016-11-29.

[CCH17] The Centre for Computing History. John von Neumann. `http://www.computinghistory.org.uk/det/3665/john-von-neumann/`, 2017. Accessed: 2017-08-21.

[Chl13] Dimitris Chloupis. Why Did the Smalltalk Programming Language Fail to Become a Popular Language? – Answer of Dimitris Chloupis. `https://goo.gl/4e1rAX`, 2013. Accessed: 2017-03-09.

[COD15] Codeanywhere Inc. Most Popular Desktop IDEs & Code Editors in 2014. `https://blog.codeanywhere.com/most-popular-ides-code-editors`, 2015. Accessed: 2016-11-29.

[Dal17] Georgi Dalakov. LISP of John McCarthy. `http://history-computer.com/ModernComputer/Software/LISP.html`, 2017. Accessed: 2017-08-21.

[Eng16a] Richard Eng. How Learning Smalltalk Can Make You a Better Developer. `https://techbeacon.com/how-learning-smalltalk-can-make-you-better-developer`, 2016. Accessed: 2017-03-10.

[Eng16b] Richard Eng. Why Smalltalk Failed to Dominate the World. `https://medium.com/smalltalk-talk/why-smalltalk-failed-to-dominate-the-world-93e7e4195039#.703zlrb72`, 2016. Accessed: 2017-03-10.

[Leo07] Ramon Leon. Why Smalltalk. `http://onsmalltalk.com/why-smalltalk`, 2007. Accessed: 2017-03-09.

[Pat13] Andy Patrizio. The History of Visual Development Environments: Imagine There's no IDEs. It's Difficult if You Try. `http://disq.us/t/hl7fs1`, 2013. Accessed: 2017-02-08.

[PHA11] Pharo Community. Announcements: an Object Dependency Framework (in *Pharo – the collaborActive book*). `http://pharo.gemtalksystems.com/book/LanguageAndLibraries/announcements/`, 2011. Accessed: 2017-04-11.

[Phi14] Lee Phillips. Scientific Computing's Future: Can Any Coding Language Top a 1950s Behemoth? (Ars Technica). `https://arstechnica.com/?p=457395`, 2014. Accessed: 2016-11-24.

[SO17] Stack Overflow. Most Loved, Dreaded, and Wanted Languages (Developer Survey Results 2017). `https://stackoverflow.com/insights/survey/2017#most-loved-dreaded-and-wanted`, 2017. Accessed: 2017-03-23.

[Swa16] Doron Swade. The Babbage Engine: The Engines (Computer History Museum). http://www.computerhistory.org/babbage/engines/, 2016. Accessed: 2016-11-22.

[TIO17] TIOBE software BV. TIOBE Index. https://www.tiobe.com/tiobe-index, 2017. Accessed: 2017-08-21.

Part VI

Appendices

Blended, not Stirred: Multi-Concern Visualization of Large Software Systems

SOFTWARE DEVELOPMENT involves a variety of activities and tools, components and environments, that relate to many different aspects of a system. Modern software development is an integration process where the developer defines a behavior by orchestrating and specializing library components and third-party entities. This has turned the engineering of any software system into an information-heavy process, which is ultimately distilled into source code. Besides source code, its structure, and architecture, development involves a large amount of corollary information (*i.e.,* discussions, design decisions, email communication between developers, bug reports, *etc.*) that is often neglected. Most of the existing visual approaches consider only single concerns, such as the architecture, the structure, the evolution, the relationships, *etc.*, but there is little in terms of visualizing multiple concerns at once.

In this chapter we present an approach, developed together with Tommaso Dal Sasso, to visualize multiple concerns concurrently by *blending* them together. Our aim is to answer one of the most often asked questions raised by developers and managers alike, namely "*what happened to our system recently?*" [SMDV08]. The concerns we tackle are interaction data, failure information, and evolution. Interaction data stems from how developers interact with the integrated development environment (IDE) while developing and maintaining a system. In essence, it provides evidence of where and how people have been active while developing [MML15b]. Failure information is generated each time the debugger is triggered because an exception has been raised. Dal Sasso *et al.* shown that such data can be leveraged to understand where the particularly tricky spots in a software system are located [DSML15]. Both interaction data and failure data are more fine-grained than their respective counterparts, namely versioning information and bug reports. We complement these two types of data with a third one, the evolution of the system. Our view uses the city metaphor to represent software systems [WL07] and employs linear color blending to portray the combination of events happening on each program entity. We present four development stories obtained through our visualization that illustrate interesting properties of an existing software project and its community.

Structure of the Chapter

Section A.1 describes the ingredients of our blended visualization, which is presented in Section A.2. Section A.3 uses the visualization to tell interesting development stories. Finally, Section A.4 summarizes and concludes our work.

A.1 The Ingredients

Our visualization blends together different data sources and enabling visual analytics from heterogeneous and multidimensional perspectives. To get a tractable subset of data, we focus on a timespan ranging from January 1st 2015 to May 1th 2015. This section describes the three main ingredients and the supporting tools that enable the data collection process. The software system under analyses is the *Pharo* IDE itself, whose code is open-source.

A.1.1 Source Code Changes

In evaluating the growth and evolution of a system researchers typically use the number of changes that a system undergoes during its development. In the case of *Pharo*, the whole system is self-contained and distributed as an *image*, a single file that works as a virtual environment where new code is installed inside the default system. The *Pharo* system is released once a year, and during this period it goes through an intense phase of improvement, debugging and polishing. The test and release process is managed by a continuous integration server,[1] that stores the previous builds of the system. In our analyses we modeled and extracted all the source code changes between subsequent releases of the *Pharo* system.

Retrieving the Different Versions. We focused on the release of *Pharo* 4, which just finished its release cycle. We downloaded all the development versions from the file server,[2] that we also used to retrieve the exact release date of each version. The full cycle of development images ranges from version 40,000 to the image 40,613, from May 26th 2014 to May, 5th 2015. The last release in date May 1th 2015 was version 40,611.

Extracting a System Model. We extracted from each image a model representation of the system. Such a model is composed of the names of all packages, classes, instance and class methods, and instance and class attributes.

Generating an Incremental Change Model. We leveraged each system model to obtain an incremental diff model that describes each change. We considered as change a variation in the names of the collected entities. Since we had no way to precisely determine when an entity was renamed, we considered every event in terms of creation and deletion. Table A.1 summarizes the available source code changes data.

Table A.1. Dataset – Source code changes in the considered period

Metric	Value
Number of Considered Versions	611
Number of Changes	4,928
Average Number of Changes per Version	8
Max Number of Changes per Version	527
Min Number of Changes per Version	0

[1] See `https://ci.inria.fr/pharo/`

[2] See `http://files.pharo.org/image/40`

A.1.2 ShoreLine Reporter and Stack Traces

A consistent part of the time spent by developers consists in finding and solving defects. The debugging activity involves tests to reproduce a problem or verify that a defect has been solved. This process generates many stack traces, that contain valuable information about the failures in a system. Such information is normally used by a developer to identify a faulty status in her program. Moreover, if collected and stacked together, stack traces can also give a hint of what parts of the system are the most active or which ones are causing more troubles. To exploit this source of information, we developed ShoreLine Reporter [DSML15], a platform to collect and store stack traces generated by the *Pharo* community. To keep track of the entities involved in the failure, we collect the method signatures of the invoked methods. We do not collect method parameters to avoid privacy concerns.

In enabling the reporter, each developer can decide to inspect each stack trace and choose the ones to submit, or enable the automatic reporting feature and submit all the traces produced by its activity. While this option produces many duplicates and non relevant data, it is still interesting to see where the activity of the developers focuses in different periods of time. The collected data can then be used to aid the debugging activity, for example detecting if a large volume of new stack traces coming from different developers involve a specific class, or by looking for existing bug reports in the bug tracker to provide a contextual help when a user encounters an exception and ease the understanding of a piece of code. The presence of many different stack traces for a specific component might also suggest that an API has a problematic design, and that the users struggle in understanding its usage, thus highlighting the need for documentation or refactoring. Table A.2 summarizes the stack traces data collected with ShoreLine Reporter.

Table A.2. Dataset – Stack traces data in the considered period

Metric	Value
Number of Traces	14,884
Number of Submitters	43
Total Number of Stack Trace Lines	714,420
Average Stack Trace Size (in Lines)	48
Longest Stack Trace	1,086
Shortest Stack Trace	1

A.1.3 DFlow and IDE Interaction Data

DFlow, extensively discussed in Chapter 5, records 32 different types of events at different levels of abstraction. For this work we only focused on a subset of meta events that involve code entities. Some meta events have an associated program entity: A browse event, for example, where the user opens a new code browser, can be performed on a method or on a class.

For this work we aggregated all meta events to the class-level: An event performed on method `foo` of class `Bar` counts as an event involving directly the class `Bar`. In total we have ca. 239,000 interaction data events covering a timespan of 4 months (*i.e.,* from January to April 2015).

The IDE interactions impact 2,988 different classes, of which 965 are part of the standard *Pharo* distribution. The remaining 2,023 classes are user defined classes that are outside the scope of our study. Out of the 32 types of meta events recorded with DFlow [MML15b], only 13 types of events appear in the dataset. This is because some of the recorded meta events do not carry any information related to program entities. For example, the meta event that represents

the opening of a `Finder`, a user interface used in *Pharo* to search for pieces of code, has no associated program entity. Table A.3 summarizes the dataset and provides additional details.

Table A.3. Dataset – IDE interaction data in the considered period

Metric	Value
Number of Interaction Events	238,741
Number of Developers	18
Number of Interested Classes (in *Pharo* distribution)	2,988 (965)
Number of Different Event Types (Total)	13 (32)

Summing Up. Our goal is to develop a visualization approach which can represent diverse data sources, such as the ones we just presented. Although we focus on these types of information, our approach can be extended to include any kind of information source related to software development.

A.2 Visualization Principles

Until now the various "ingredients" of software development have bee processed and visualized in isolation, leading to an incomplete view of the system. Our goal is to visualize all these ingredient to enable a quick and comprehensive assessment of what happened to a software system in a given time frame. To do so, we propose the "*Blended City*", a visualization that uses the *City Metaphor* [WL07] to depict all the ingredients of a software system. Figure A.1 shows the tool we implemented to visualize the Blended City.

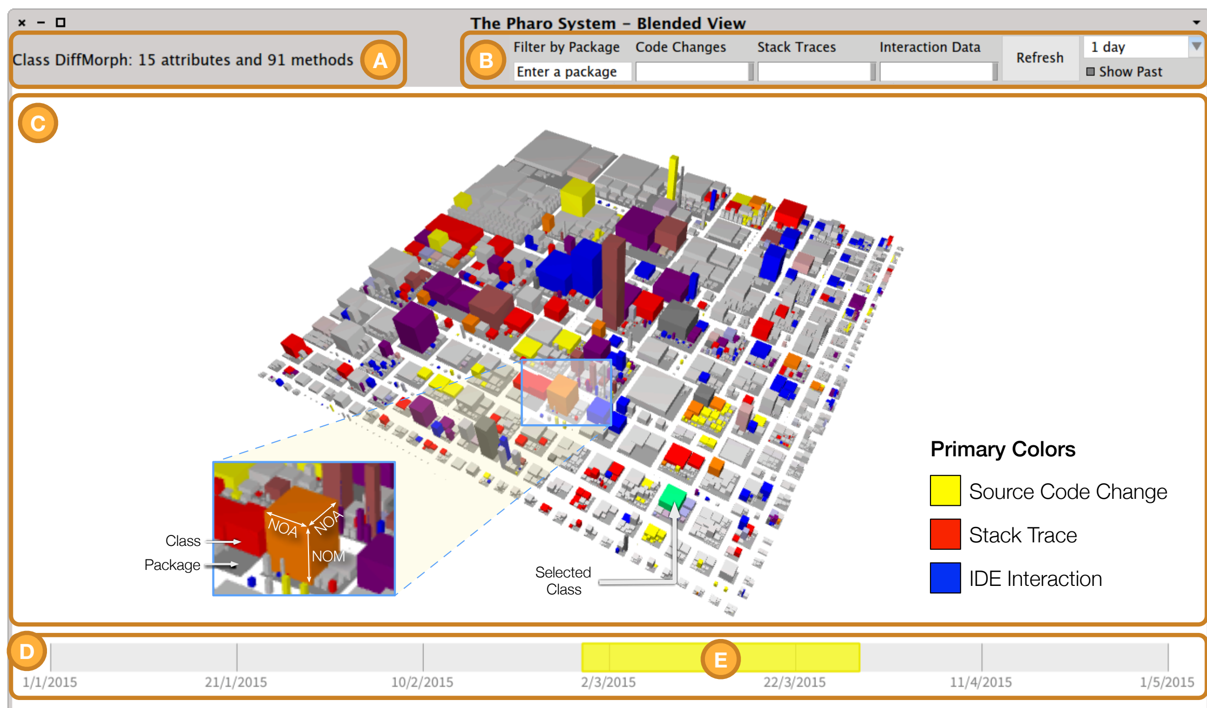


Figure A.1. The Blended City – Visualization principles and proportions

The tool is composed of four main parts: (A) A status bar to display additional information on the selected entity, (B) a toolbar to customize the visualization, (C) the view canvas, and (D) a timeline slider. With the timeline slider the user chooses the visualized data timespan. The width (*i.e.,* granularity) of this slider can be adapted using the dropdown menu on the right part of the toolbar. In the example of Figure A.1 the user selected one month of data, starting from March 1st. The toolbar (Figure A.1.B) also features a text-input and a set of sliders. The former enables simple queries to highlight particular packages in the system while the latter let the user choose the visual weight of each of the three ingredients of our visualization. These weights affect the intensity of the color associated to each of the ingredients. In the example of Figure A.1, all the sliders are at 100%, thus all the ingredients have the same importance.

Figure A.2, instead, shows the same data of Figure A.1 giving high importance to stack traces (100%), little importance on interaction data (50%), and no importance to source code changes.

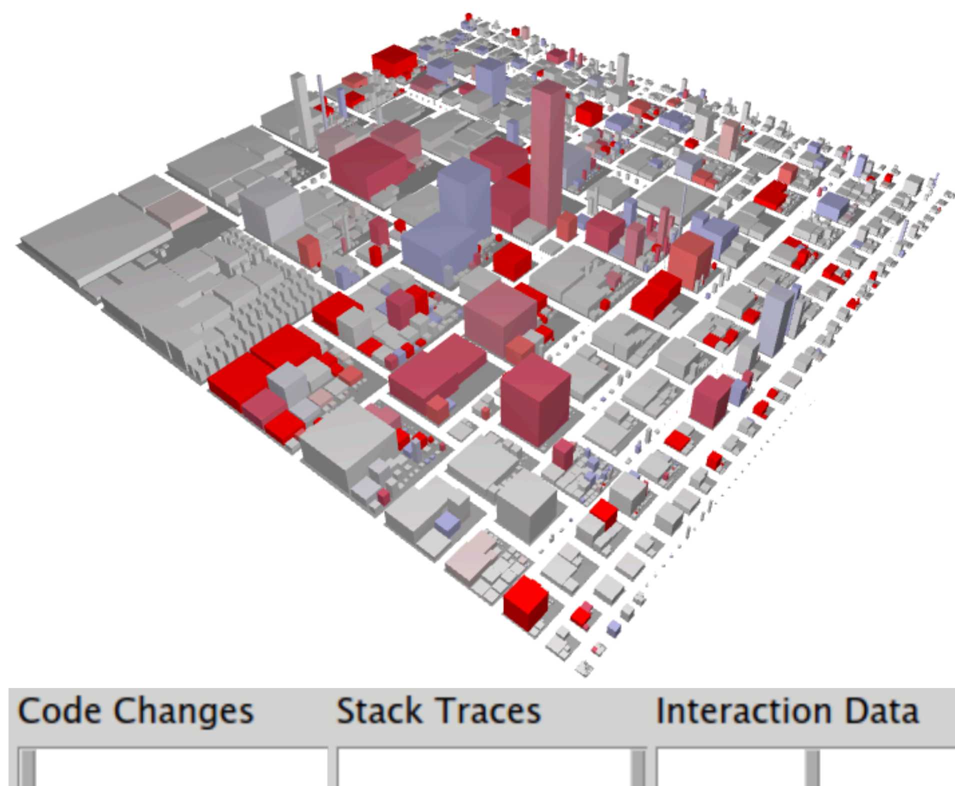


Figure A.2. The same view of Figure A.1 with different ingredients (0% - 100% - 50%)

In addition to changing the weights of the three components and the granularity of the visualized timespan, the view also features standard interactions such as panning and rotation in the 3D space. Moreover, the user can click on an entity and get additional information on the status bar. In the example depicted in Figure A.1 the user selected the class `DiffMorph` and the tool shows that this class has 15 attributes and 91 methods (see Figure A.1.A). Selected entities are colored with a bright green.

A.2.1 The City Metaphor: Layout and Metrics

In the city metaphor every district of the city is a package and the buildings, contained inside the districts, represent the classes [WL07]. The view uses a rectangle-packing algorithm to create the layout and it is *polymetric*, *i.e.,* each dimension of the visual entity is proportional to a particular metric of the program entity being represented [LD03]. Since the visualization is 3D, classes are cuboids and have 3 dimensions that correspond to three metrics. Our visualization, similar to the original CodeCity, uses the same metric for both width and depth and a different measure for the height. In particular, we use number of attributes (*i.e.,* NOA) for both width and depth of a class and number of methods (*i.e.,* NOM) for the height of the cuboid representing a class. The magnification in Figure A.1 exemplifies these mappings.

A.2.2 Color Harmonies and Blends

Our Blended City presents different types of data, from structural properties of source code to stack traces and interaction data. Structural source code relationships (*i.e.,* nesting of the package and software metrics) are the foundations for the layout while colors present the remaining information. We use a triadic color scheme made of primary colors to present this information: Yellow for source code changes, red for stack traces, and blue for interaction data. Figure A.3 shows a the color wheel with an emphasis on the triadic color scheme, where colors are evenly spaced around the color wheel.

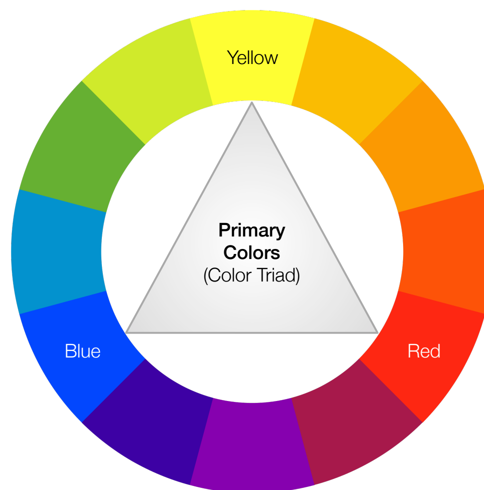


Figure A.3. Color wheel and triadic color scheme

This offers strong visual contrast while retaining balance, and color richness. Using colors equally spaced around the color wheel facilitate the addition of extra sources of information, *i.e.,* when we need to display n sources of information, we can create a new color harmony composed of n colors evenly spaced around the color wheel.

Color Blends. The three primary colors can only depict entities which are affected by a single of the three information sources. However, in a given timespan a class might be affected by both IDE interactions and stack traces, for example when a developer is adding new functionalities to a class and testing them. To depict this information, we use linear color blends between the different sources of information. A class with both IDE interactions and stack traces is depicted in purple, the linear blend between the color of IDE interactions (*i.e.,* blue) and stack traces (*i.e.,*

red). Figure A.4 shows examples of the different linear color blends on the triadic color scheme adopted by our visualization. In this work we only considered the linear blending of colors. It is part of our future work the investigation of different techniques to combine the colors, *i.e.*, color-weaving.

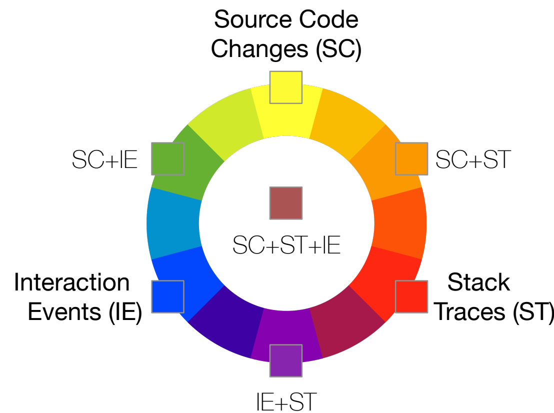


Figure A.4. Linear color blend on triadic color scheme

Aging Mechanism. When the user selects a timespan to visualize, the tool pre-loads and displays also the data happening in the immediately preceding interval (of the same length). This enables the user to draw conclusions from the visualization having also in mind what happened immediately before. To show this data, the tool uses an *aging mechanism* that linearly reduces the color saturation as the age of the datapoint grows, *i.e.*, the older the more intense fading towards the default color of nodes (*i.e.*, gray). Figure A.5 shows how colors fade with such mechanism in a timeline.

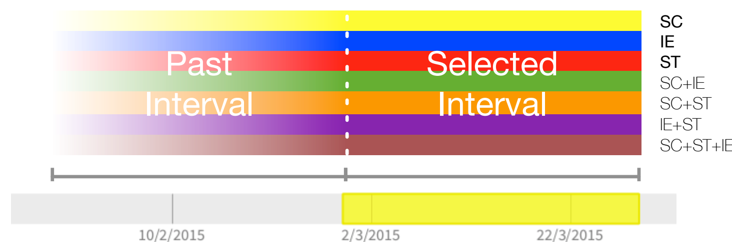


Figure A.5. Aging process: example in the Timeline

In the "present" interval (*i.e.*, the one selected by the user), colors are at their default saturation. In the "past" interval, instead, the color saturation fades. At the end of this interval, the nodes have the default color, *i.e.*, light gray.

A.2.3 Under the Hood

The tool deals with a large volume of entries coming from heterogeneous data sources. To conveniently manage them we standardized their format, using different data pre-processors, and store them in a central place. We use *MongoDB*[3] databases to conveniently store the data.

[3] See `http://mongodb.org/`

When the user selects a timespan to visualize (Fig. A.6.1), the tool loads the data through optimized *MongoDB* queries (Fig. A.6.2) and builds the blended model of the data (Fig. A.6.3). Later it computes the city layout, applies the blended color scheme, and presents the view to the user (Fig. A.6.4). The user can then use the toolbar to refine the visualization (Fig. A.6.5).

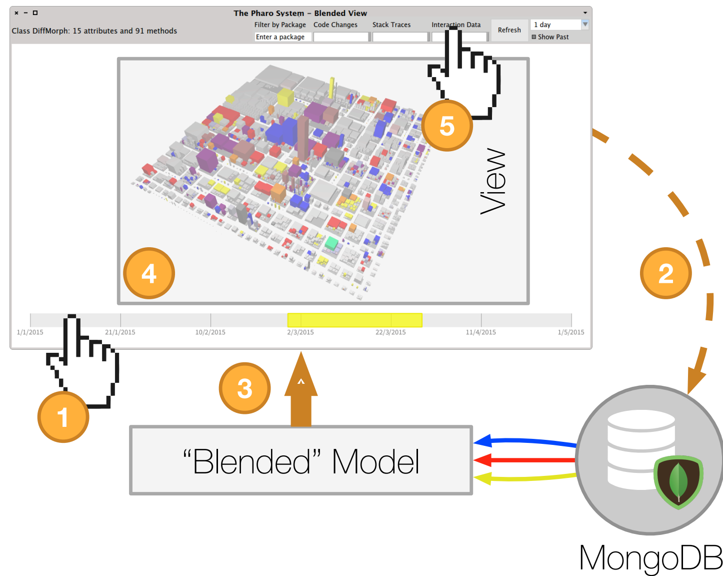


Figure A.6. The architecture of the Blended City

A.3 Telling Development Stories with the Visualization

This section presents four stories,[4] that narrate the evolution of the *Pharo* system:

- Those Awkward Neighbors;
- Market Districts;
- New in Town; and
- The Purple Buildings.

[4]To increase the readability of this Section, we present each story on a new page.

Those Awkward Neighbors

By selecting the full available timespan of the data we obtain a visualization that displays all the activities that involved the *Pharo* system over a period of five months. This enables to obtain a comprehensive view of the system evolution and derive long-term considerations and properties. Figure A.7 shows the overall view of the available data. One interesting example is represented by what we call the *awkward neighbors*, *i.e.,* big but silent packages that have little or no activity.

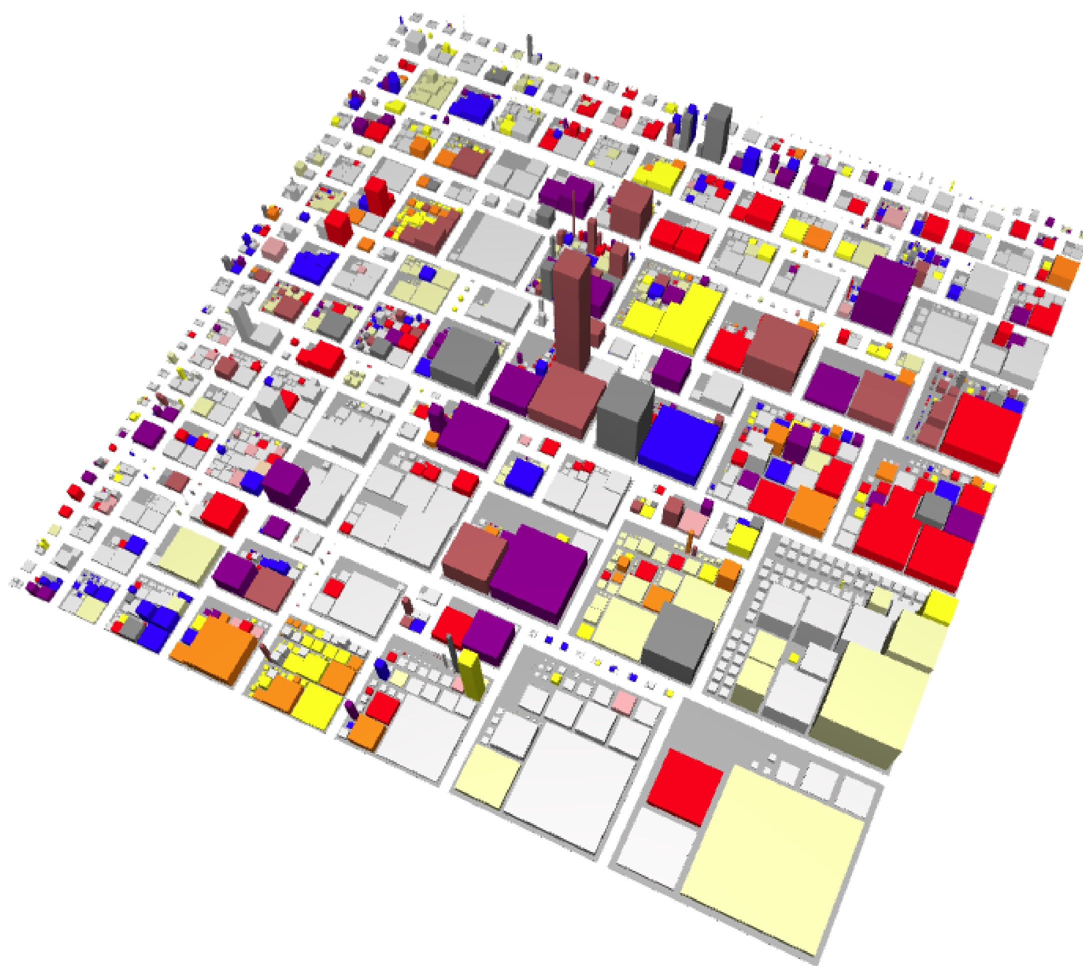


Figure A.7. View of the City with all the Activities

In the lower part of Figure A.7, we can spot two big packages that contain entities that are mostly colored with grey, meaning that they had almost no activity in the whole timeframe. Moreover, they present almost no change in the entities they are composed of, and since the color of the changes is blended, those are all antecedent to the selected start date. This means that in the last release they have been mostly ignored. These two districts are the packages *Graphics-Files* and *Compiler*.

A further investigation of the package *Graphics-Files* reveals that it contains 10 classes. These classes are dedicated to exporting graphics and writing them in different file formats. Since *Pharo* stores the dates of the changes of a method, we can determine when the changes took place. We can see that there are three main batches of changes: A small update in 2014, regarding a small

refactoring of an error message, one in 2001 and one in 1997. This is interesting, because it indicates that the package has been part of the system for a long time, it had little changes and is by now a solid foundation of the system. Similarly, the package *Compiler* contains 46 classes, and apart from some recent modification in 2013 to the structure of the compiler, many of the methods are unmodified since 2006, 2003, or 1998.

One might wonder how it is possible that some parts are older than the *Pharo* project itself. The reason is that *Pharo* was born as a fork of the SQUEAK project[5], which in turn is a re-implementation of the original SMALLTALK-80 system, which was evolved from the SMALLTALK-72 system. This means that some of the methods and classes in these packages might very well be 40+ years old.

[5] See `http://www.squeak.org`

Market Districts

While examining some of the packages with the most activities, we found districts with many interactions from all three data sources, and we call them *market districts*. Figure A.8 shows an example of market districts corresponding to the packages of *Spec* and *Morphic*. *Morphic* is the core graphic library of *Pharo*, while *Spec* is an UI framework built on top of *Morphic*.

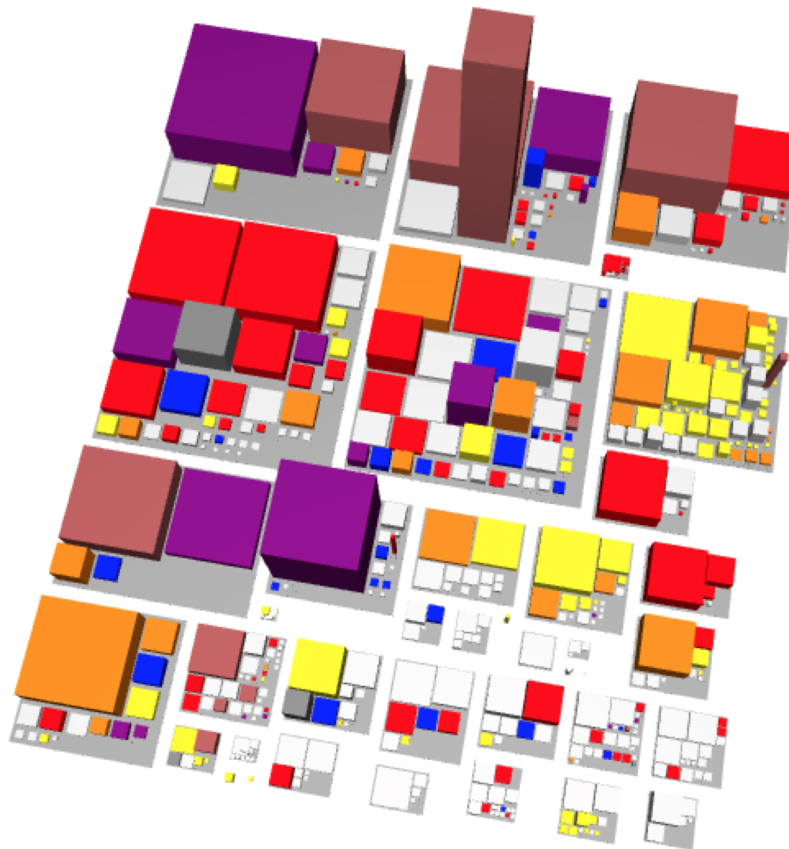


Figure A.8. *Spec* and *Morphic* Market Districts

Many classes are involved in exceptions, they were recently changed or they were subject to developer interactions. This reveals a long known problem in the community: The code of *Morphic* is old and have been ported through various platforms. The case of *Spec* is similar: since *Spec* is a framework built on top of *Morphic*, it shares its weakness and part of its complexities.

Differently from the awkward neighbors, the market districts for *Morphic* and *Spec* are not settled and solid: Instead, they are often causes of bugs and issues. The view also shows that many classes that act as entry points received frequent developer interactions, meaning that they likely have an unclear public interface.

Moreover, we can see that the *Morphic* packages are still frequently changed, showing that the community is constantly trying to fix the codebase. Finally, the high number of classes involved in the stack traces suggests that the code modification, together with the difficulty of understanding the API, is likely a cause of many programming errors. In particular, there are some *hotspots*, *i.e.,* packages where classes are mostly colored in red only. These classes are involved in failures, but they are rarely modified or involved in interaction data.

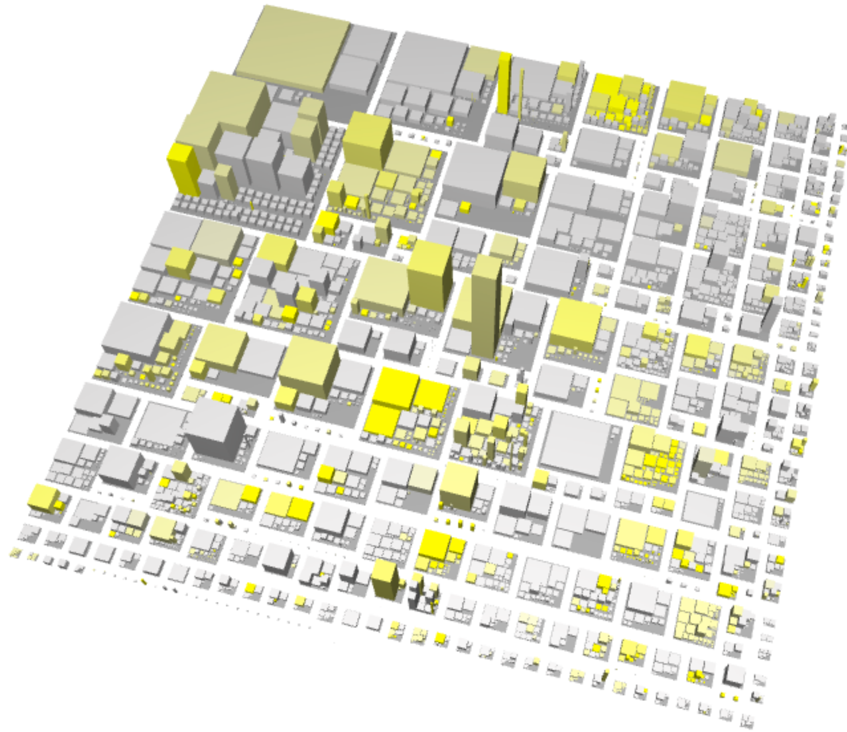


Figure A.9. Changes in the *Pharo* system

These theses are confirmed by the fact that the community is trying to replace the code of *Morphic* with a new, polished and easy-to-use replacement called *Bloc*, to address issues that we can be spot in Figure A.8. However, as the complexity of the picture suggests, replacing this code is not an easy task, and has been work-in-progress for more than a year now.

New in Town

During the development of *Pharo* 4, many classes got updated and some new components were added. We want to analyze the progressive introduction of these changes, and how they impacted the system after the integration. We then use the sliders in Fig. A.1.B to remove all the data sources, except for the changes. Figure A.9 shows in full yellow the entities touched by a change in the last five months, and in blended yellow the changes in the previous five months. We can verify that there are elements that remained untouched, while some others were subject to intense development.

By moving the slider we can select a timespan to restrict the changes to a given moment of the story of the components and inspect the status of the system during time. We can notice that from a certain point on there was the appearance of packages related to the *GT-Tools*, a set of tools to improve the interaction with the objects in the system. By restricting the timespan to the beginning of January (*i.e.,* the first appearance of activities), to determine the moment of integration.

Figure A.10 visualizes the Blended City for the *GT-Tools* packages. Some classes are involved in all three data sources, *i.e.,* they are colored in dark brown. This can be explained by the fact that the first phases of integration usually require adaptation, refinement, and debugging, thus generating (other than changes) frequent exceptions and developer interactions.

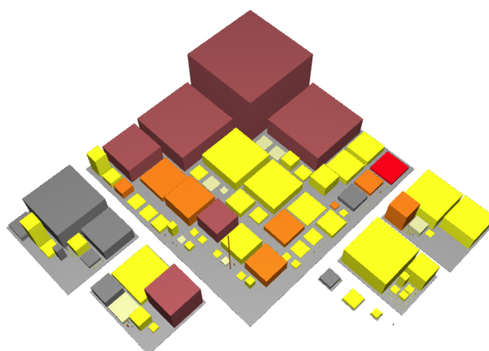


Figure A.10. The changes of GT-Tools Packages

The other interesting observation that we can derive from the visualization is that the classes involved in user activities are also the biggest. This can be explained by considering that those classes act as main entry points to the package, a starting point for developers who want to use or inspect the code.

The Purple Buildings

A benefit of our approach is that a data source can be removed to spot behaviors that are independent from it. Figure A.11 shows the Blended City for *Pharo* with only stack traces and developer interactions. While the stories we presented so far try to consider the code entities at a package level, without changes the Blended City reveals the interesting role of some classes.

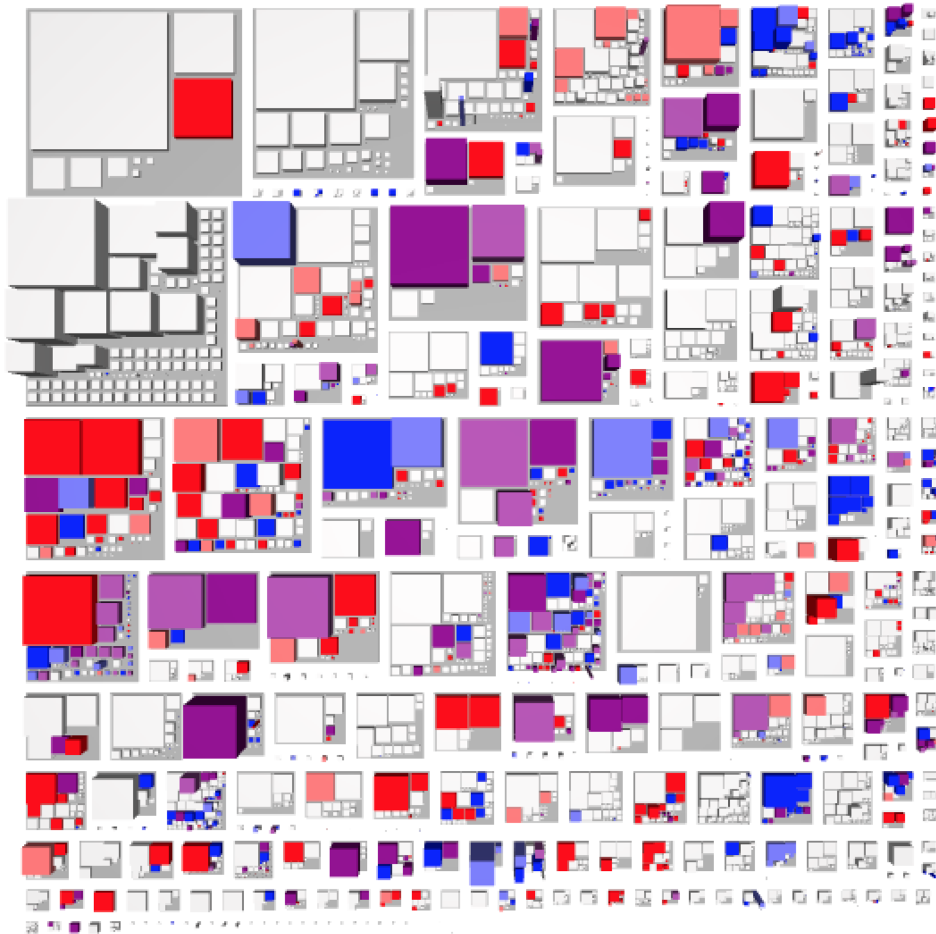


Figure A.11. A view of the system highlighting only stack traces and developer interactions

Scattered across the system, there are some big and medium classes colored of purple without apparent correlation with the color of its neighbors. By inspecting their names, we find examples like *DateAndTime*, *Float*, *Job*, *SmalltalkImage*, *Socket*, *SocketStream*, *SystemWindow*, *TestRunner*, and many others related to rendering of graphics, that we covered in the previous stories. These classes are not problematic *per se*, but represent an interesting area of the system that we could define as *Advanced APIs*. These classes appear in many stack traces and in many development interactions, an information that suggests that they occour near the source of the exceptions, when these exceptions are not directly generated from them. This context could signify that the user is trying to understand a class that has a name suggesting a behavior, but that she needs some further understanding to learn how to use the objects of the class by trying the various methods. The use of this information could be used by the maintainer of the system to prioritize the areas of that could need more public documentation, to ease the learning process of those entities and their API.

Note that the same information, blended with the addition of code changes and applied to classes that are not part of the core system, could signify that a developer is applying a *Test Driven Development* approach, by implementing incomplete methods and completing them whenever the system tries to execute a method that is not yet implemented.

A.4 Reflections

Software analyses and visualizations usually focus on giving a detailed representation of a single aspect of the examined entities. We presented an approach where we visualize data from three different data sources and contexts, blending them to produce multi-dimensional information about a system, its code and how developers interact with it. We considered a combination of system changes during the development phase of a system, the interaction data generated by users and the stack traces of the exceptions triggered during the daily usage of the platform. Out tool visualizes blended information on a city view of the source code of the *Pharo* IDE, a dynamic, flexible and active programming ecosystem. We showed how our tool allows to select different timespans and weigh the diverse components, to enable a fine grained inspection of each entity during its recent evolution.

We believe that our approach has a real potential to be successfully applied in a development context to allow for multi-dimensional incremental and interactive analysis of a system, supporting a deeper understanding of the code entities by highlighting the synergies among its recorded activities, and the relations and interesting behaviors otherwise hidden or harder to detect.

We illustrated four stories where we extract and analyze some real-world issues by looking at the blending of the data and identifying some existing problems, or finding suggestions for problems that could be addressed by the maintainers of the platform to improve the system.

Gamifying Software Engineering with Interaction Data

WHAT IS A GAME? According to McGonigal [McG11] games share four defining traits: a *goal*, *rules*, a *feedback system*, and *voluntary participation*. The goal gives a sense of purpose. The rules unleash creativity and foster strategic thinking. The feedback system provides motivation. The voluntary participation makes the experience safe and pleasurable. Suits sums it up with "*playing a game is the voluntary attempt to overcome unnecessary obstacles*" [Sui05]. McGonigal provides several examples of contexts, ranging from house holding chores to physical exercise, where the performance of subjects has been boosted through gamification [McG11]. While this may seem remote from software engineering, Werbach and Hunter provide an illuminating example closer to our discipline: *Microsoft*'s testing team in charge of the multi-language aspect of WINDOWS 7 invented the *Language Quality Game*, recruiting thousands of participants who reviewed over half a million dialog boxes, logging 6,700 bug reports, resulting in hundreds of fixes [WH12]. Another example is *StackOverflow*, a Q&A website where asking and answering technical questions is rewarded with points and badges. There is evidence that gamification is in part responsible for *StackOverflow*'s success [VFS13].

> *"The programmer, like the poet, works only slightly removed from thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination."*
>
> — FREDERICK P. BROOKS [BRO95]

This chapter describes our vision on how to use fine-grained interaction data recorded with DFLOW to develop a gamification layer on top of an IDE, the main vehicle for creating software. Modern IDEs have become powerful tool suites that allow one to construct, understand, and modify software systems. Several decades ago Weinberg defined programming as a "*kind of writing*" [of source code] [Wei85]. We believe this is a fundamentally flawed perception, and among others is also responsible for the wrong assumption that productivity can be measured in terms of lines of code [Jon78, Bro95]. We believe developers are unsung heroes and deserve a mechanism that rewards them for a good job. Programming is more than lines of code.

Structure of the Chapter

Section B.1 explains our vision and Section B.2 details our first steps. Section B.3 discusses potential directions for the future and Section B.4 concludes the chapter.

B.1 A Gamification Layer for the IDE

As opposed to the common belief that gamification is a recent, under-explored and thus not very scientific domain, behind it there is in fact a strong scientific intuition, rooted in the realm of psychology, and more specifically behaviorism, an approach that combines elements of philosophy, methodology, and theory [Ski78]. In fact, behaviorism, whose main tenet is "*if you do this you'll get that*", is the antithesis of successful gamification, because simple rewarding mechanisms like token programs have been shown to be bound to fail [Koh93] in the long run. Put simply, our goal is not to assign points to development actions. Such a simplistic approach is destined to fail. Rather, we propose a comprehensive approach where the ultimate goal is the creation of an *alter ego* of a developer, which we believe is the key to enable what McGonigal [McG11] identified as the 4 key aspects of successful gamification: i) Satisfying work (after all, programming is creative), ii) the experience/hope of being successful, iii) a social connection, and iv) a deeper meaning. We believe that the creation of an alter ego is the key to provide both short-term and long-term gratification to developers.

B.1.1 Our vision

Our aims is to devise an approach that leverages fine-grained interaction data mined from an IDE to provide a mechanism that rewards them for a good job. DFLOW models and silently harvests the low-level action performed by developers inside the IDE offering a complete and precise summary of what is being done. We identify three building blocks in our vision:

Session Digest. The session digest is a short-term form of gratification, similar to the one present in fitness apps, offered to developers for their last development session. It summarizes the last session from various perspectives, *e.g.,* how was time used, how much was achieved from a coding point of view, which program entities were involved, etc. It also enables to dig into the fine-grained recorded data and acquire a deeper understanding.

Alter Ego. A developer is like a character in a role-playing game: She moves her first steps, evolves, acquires new skills, and unlocks new achievements. Developers are thus assigned an avatar that they can evolve, providing them short- and long-term satisfactions to turn software development into a more engaging activity.

Development Empire. The last, and most ambitious goal is provide developers with long-term gratification mechanisms. We envision a comprehensive gamification layer on top of the IDE: the *Development Empire*. It is not all about assigning points to them, but a ramified system that rewards complex actions and best practices (adherence to design patterns and design heuristics [Rie96], test-driven development, etc.) of a developer with badges, achievements, and trophies of different types. The history and the evolution of the alter ego of a developer is a key factor. When this mechanism is in place, all the alter egos will originate a new community, where people can observe, challenge, and interact with other developers.

B.2 Session Digest: Free Hugs for Developers

The *"session digest"* is a form of short-term gratification for developers, that can potentially augment their level of engagement. We shape and frame the rewards in a digest as visual summary

of the development session. The digest is composed of three main parts: i) an overview of the development session; ii) a selection of fine grained information to highlight the most important actions; and iii) a glimpse on the "profile" of the developer. The general overview, *e.g.,* in terms of how the developer spent her time, serves as an entry point for the digest. Once the developer gets the global picture, she can use the remaining part of the digest to retrospectively analyze *how well she did* in the current session. The last part, the developer profile, summarizes the developer's avatar status.

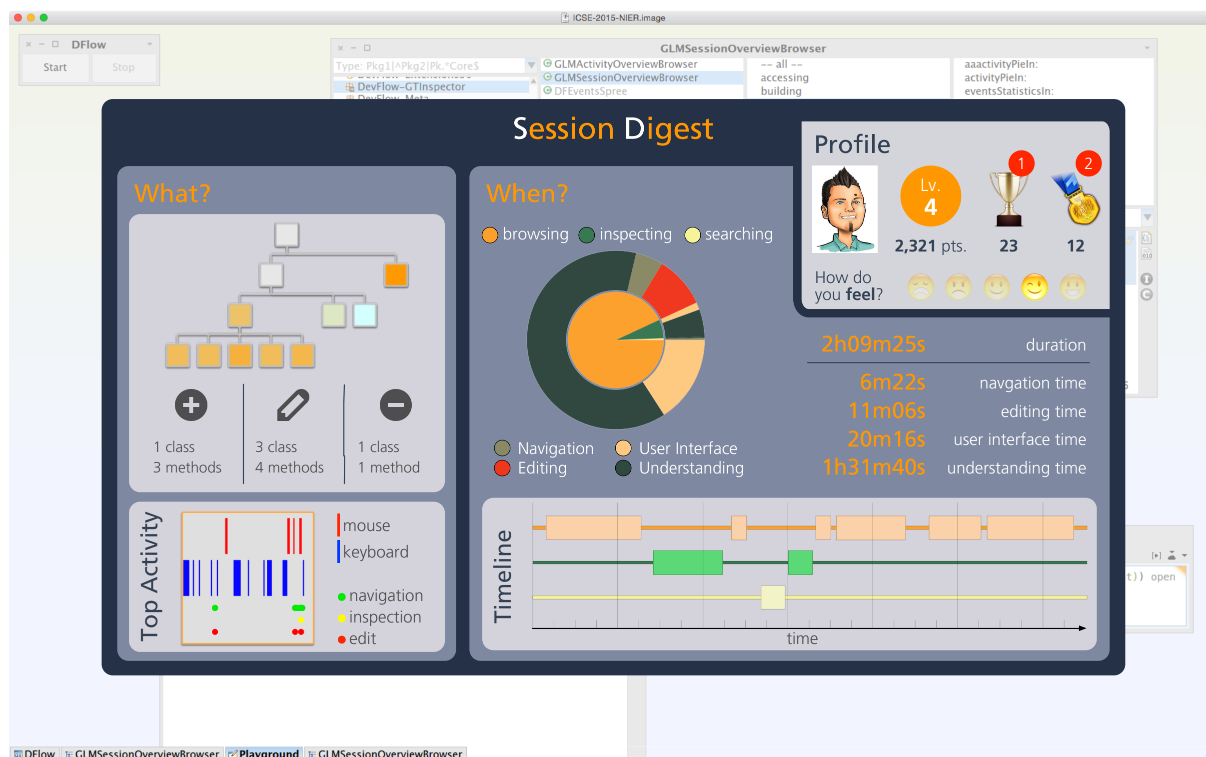


Figure B.1. Session Digest: How have you spent your time? What did you do?

Figure B.1 depicts a sample session digest prototype on top of the standard IDE window. The left part of the digest serves as a general overview on how the developer spent her time. The overview is achieved by a sunburst visualization accompanied by a set of time metrics. Figure B.2 shows the sunburst in details. Its central part distinguishes the time devoted to the three different high-level activities, namely: browsing (orange), inspecting (green), and search (yellow). For each of the three types of activities, the visualization shows four time components: navigation (green), editing (red), understanding (dark green), and user interface (pale orange).

For completeness, the visualization provides the same information as text. We quantify these time components using interaction histories mined with DFlow [MMLK14].

After an overview, a developer has the possibility to dig into her last development session by means of the central part of the digest. In the example of Figure B.1, the central part shows an interactive tree visualization that portrays all the entities that she has interacted with in her last session. The tree has 4 different levels, (1) subsystems, (2) packages, (3) classes, and (4) methods. Each entity have a color to represent the intensity of the interactions: gray for entities with no interactions (*i.e.,* inserted only as a transitive closure to complete the tree) and a color scale from light blue (*i.e.,* few interactions) to orange (*i.e.,* lot of interactions) for the other elements. Below this view there is a table that shows how many entities were respectively added,

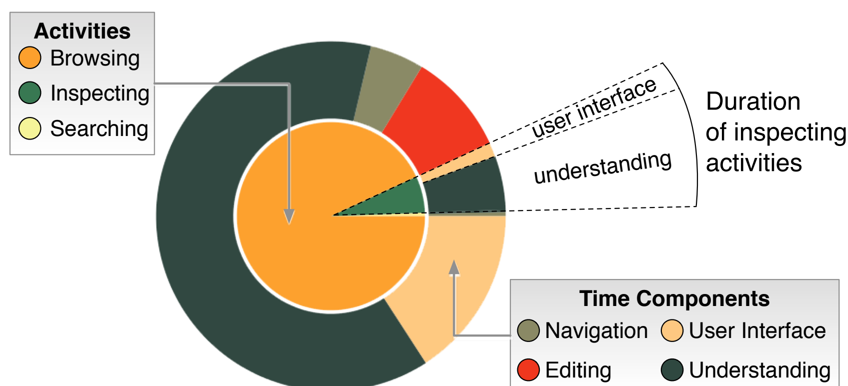


Figure B.2. Activities and time components in a sunburst visualization

modified, or removed. In Figure B.1, the bottom right corner presents the *activity view*. This view, detailed in Figure B.3, uses a custom layout to decompose a single high-level development activity. It depicts mouse, keyboard, and *meta* events. The first two kinds of events have a duration, proportional to the width of the rectangles representing them. *Meta* events have no duration since they represent IDE actions such as *saving a method*. Their color represents their type, according to the impact they have on source code. Navigation events (green) do not modify source code, inspect events (yellow) are a deeper form of navigation (*i.e.,* in a debugger), while editing events (red) modify source code. In the session digest, for example, we can show the *hardest activity*, as depicted in Figure B.1.

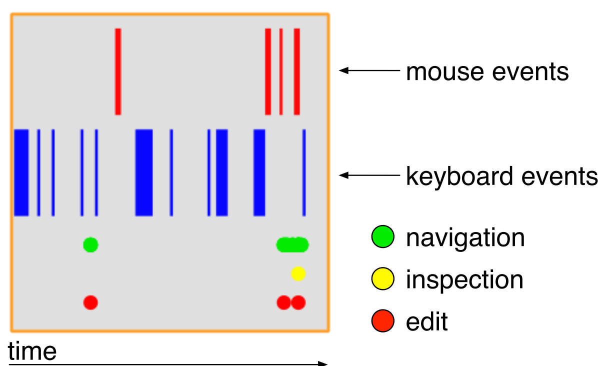


Figure B.3. Activity View: Decomposing an high-level development activity

The last component of the digest is the developer's profile, depicted in the top right corner. It shows a profile picture of the developer, her level, points, trophies, badges, and lets her provide a "sentiment feedback" about the last development session on a *smiley scale*. This enables analyses on what characterizes, for example, a frustrating session. Figure B.1 shows a prototypical profile of the first author of this paper. He is at level 4, with 2,321 points. In the last session he achieved 1 new trophy and received 2 new badges. Overall, he owns 23 trophies and 12 badges. This goes towards an application of gamification in software development [Mas14]. Gamification, if carefully engineered [McG11], can increase the motivation and engagement of developers. For an open source community, such as the one behind the *Pharo* IDE[1], our target development environment, this can provide several benefits. Having more motivated people will most likely

[1] See `pharo.org`

increase both the quality and the quantity of the contributions to the community. At the same time, this will originate a novel developer community that, at first sight, is similar to Open HUB,[2] the open source network formerly known as Ohloh. But our vision goes beyond *mere* version control system data. We imagine a rich developer profile that includes interactions with different sources of information including, but not limited to, IDE interactions. For example, the system will integrate bug tracking systems, questions and answers services, mailing list participation, etc. Points, trophies, and badges will cross the boundaries of semantically different domains, delineating a comprehensive profile for developers.

B.3 Extending the Session Digest

Regardless from the fact that we can present IDE interaction data from different perspectives, the session digest can also be extended in multiple, orthogonal, directions. When we realize the vision discussed in Section 14.1, the digest could leverage and present information coming from different sources. If we only focus on the research topics of the members of our research group,[3] at the time of writing, we foresee the following extensions:

Bug-tracking Systems. Tommaso Dal Sasso is working on how the process of submitting bug reports and patches can be ameliorated [SL14]. Dal Sasso *et al.* are trying integrate the process of submitting a bug report directly within the IDE. Once bug-tracking information is available in the IDE, statistics about how a developer behaves in reporting and fixing bugs can be integrated inside the session digest. For example, we can keep track on *how active* a contributor is in the bug-fixing process, *i.e.,* how much she contributes, how many bugs she fixes, or how many issues she submits to the bug tracker system.

Coding Style and Guidelines. Yuriy Tymchuk[4] is working on code style and quality metrics. The session digest can include evolutionary visualizations on how quality and style of a software system evolved over time. This information gives a tangibile feedback, and intrinsically a reward, to developers investing efforts in ameliorating the design and implementation of their systems.

Questions & Answers Services (Q&A). Luca Ponzanelli is working on integrating Q&A services inside the IDE. We envision IDEs that harness the potential of the crowd knowledge, for example by letting a developer read, answer, and post new questions on these platforms directly from the IDE [PBL13]. At that point, our digest can include such information and reward developers that better exploit and contribute to the crowd knowledge base.

Discussion

These ideas are only few of the many sources of data that the session digest can present to a developer. In turn, they set the ground for our more ambitious and extensive gamification layer built into an object-oriented IDE.

[2]See `https://www.openhub.net`
[3]See `http://reveal.inf.usi.ch`
[4]Former member of REVEAL, now part of the SCG Group at the University of Bern.

B.3.1 How Can We Evaluate a Gamification System?

A good starting point to evaluate a gamification system is to assess to what extent it meets McGonigal's four key aspects of successful gamification [McG11]:

1. *Satisfying work.* In his book Graham compares programmers (or better hackers) to painters, saying that programming is a creative activity not dissimilar from painting [Gra08]. According to McGonigal, creative work is one of the kinds of work that leads to satisfaction. We should assess to what extent our gamification system makes programming an even more satisfying work.

2. *Experience/hope of being successful.* We should assess to what extent short- and long-term rewards offered by the Development Empire are enough to keep high the hope of being successful.

3. *Social connection.* The participation of programmers to the Development Empire will originate a novel community. The hope is that community promotes new social connections and is an additional source,

4. *Deeper meaning.* The final aim is to understand whether the Development Empire serves to grant a deeper meaning to software development activities.

Our target IDE is always the *Pharo* IDE. Behind *Pharo* there is a an open-minded and reachable community, with whom we are directly in touch. If we introduce such a gamification layer inside the *Pharo* IDE we should rely on *Pharo* developers and strongly value their feedback.

The goal while evaluating such a system is to assess to what extent developers benefit from what we propose, in terms of engagement, self satisfaction, and improved productivity. The first step is to release the gamification system to the *Pharo* community and conduct a continuous qualitative evaluation. Interviews and questionnaires will help us to evaluate the design of the gamification system and refine it to meet the expectations of developers. This would enable a positive feedback loop with developers, eliciting latent issues that cannot be foreseen in advance. Another hypothesis would be to conduct a comparative evaluation between developers participating in the *Development Empire* and developers using the standard *Pharo* release. We want to assess different parameters such as satisfaction, engagement, productivity, and quality of the code being produced. We imagine that developers involved in the *Development Empire* will likely achieve better results, in terms of code quality, productivity, and community engagement with respect to the others. At the same time they might "feel better" due to the rewarding mechanism offered by our system.

B.4 Reflections

We presented our vision and initial design of a micro-gamification layer on top of an object-oriented IDE. Our long term goal is a system that rewards long-term growth in terms of development skills. Synergies between gamification and software engineering are a very novel phenomenon. In this work we first explained the scientific intuitions, which are rooted into in the realm of psychology, a field which is orthogonal to software engineering. We strongly believe that such an approach might have a positive impact, particularly—but not only—on small and open-minded communities, such as the one we are targeting.