Visuo-spatial Programming:
Visualising software structure to aid the integration of egocentric software navigation into an allocentric spatial cognitive map that supports software navigation and composition
Daniel Robert Bradley
BInfTech (Honours), GDipMolBiol, GCResComm

A thesis submitted for the degree of Doctor of Philosophy at
The University of Queensland in 2016
School of Information Technology and Electrical Engineering
Abstract

Despite extensive research into the cognitive processes used by programmers to form a functional mental model of software during program comprehension, there has been little research into how the structure of software is represented within long-term spatial memory. It is conjectured that this lack of emphasis on the spatial aspects of code has inadvertently resulted in mainstream software development environments not adequately supporting relative navigation of the software call graph, which results in programmer disorientation.

While software understanding tools for visualising object-oriented software have been developed that leverage spatial memory, opening a class to reveal its source code usually obscures the spatial representation and also places the source code in a single common location that has no spatial relationship to the code just navigated from. This is likely to interfere with the integration of spatial information related to individual source code files into a common cognitive map within spatial memory.

A key challenge that tool designers face is that, for any non-trivial program, it is impossible to represent all of the source code of a program on the screen at once. Recent prototype environments have used a variety of strategies, including using a semantic zoom that allows classes to be represented with differing amounts of information visible, allowing fragments of classes to be arranged on independent surfaces, and representing individual methods as bubbles that can be grouped within a scrollable workspace.

This project has taken a different approach by conferring a spatial structure on source code that is based on its emergent structure, and implementing a visualisation technique that uses this structure to provide a consistent spatial representation of methods. The method-flow visualisation technique has been developed to support short-term spatial memory by placing editor columns within a scrollable flow view that ensures each column maintains a consistent spatial position if scrolled. As the programmer navigates a call graph by following hyperlink-enabled method calls, editor columns are added to the right-hand end of the flow view. At any time, the programmer can scroll the flow view to see previously traversed methods. It is theorised that method-flow allows short-term visuo-spatial memory to be refreshed, and provides more time for methods to be integrated into a cognitive map within spatial memory.

Method-flow has been implemented as the Visuocode prototype software development environment and has been both informally and formally evaluated. Formal evaluation
consisted of two qualitative think-aloud studies — the first of software navigation, and the second of program composition. Both formal studies asked each participant to perform a number of programming tasks using both Eclipse and Visuocode. During the studies, the screen activity of participants was recorded using screen-capture software, and later analysed to determine the number and type of navigations performed during each session. During both studies, participants performed more relative navigations while using Visuocode than while using Eclipse, and performed more direct and scrolling navigations while using Eclipse than while using Visuocode. While using Eclipse, participants tended to open classes, and then scroll down through them to discover class interrelationships – forming a mental model based on the program’s composition hierarchy. In contrast, while using Visuocode, participants tended to open the outline of a class, and then use method-flow to traverse the call graph to discover how classes interrelated – forming a mental model based on the program’s call graph. Further research is required to determine the advantages and disadvantages of either form of mental model.

It is concluded that the method-flow visualisation technique is both intuitive to use, and also useful, as all participants used method-flow extensively during at least one activity. Method-flow was extensively used to navigate code during the software navigation study, and during such navigation no participants were observed to become disoriented while backtracking. During the program composition study, although method-flow was mainly used to refer to Javadoc documentation, Visuocode functioned as an effective environment for composition — two out of the three participants who successfully completed a task did so using Visuocode.

One issue with method-flow was identified. While using Visuocode during traversal of the call graph, participants are often not made aware of other methods that are contained within any classes traversed — analogous to a horse that is ‘blinkered’. After their sessions, one participant expressed feeling misled by method-flow, and another mentioned that it took a while for them to get used to not looking at whole source files. In conclusion, while method-flow is considered to be a promising technique for navigating software, the effects caused by being ‘blinkered’ need to better understood.
**Declaration by author**

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.
Publications during candidature

Publications included in this thesis
No publications included

Contributions by others to the thesis
No contributions by others

Statement of parts of the thesis submitted to qualify for the award of another degree
None
Acknowledgements

I would like to thank the University of Queensland for providing me the opportunity and resources to study for a PhD. During my candidature the University allowed me to enrol in and complete a Graduate Certificate in Research Commercialisation, and also provided me with an associated financial grant that I was very grateful for. I would like to thank all the staff and students who provided me with friendship and support during my time at the School of Information Technology and Electrical Engineering. Specifically, I would like to thank the people below, without whom I would not have been able to complete this thesis.

I would like to dedicate this thesis to my first primary advisor David Carrington, who originally supported my application as a PhD student, but who sadly passed away during my candidature.

I would like to express my thanks to Ralf Muhlberger who took me on as a then part-time research student and gave me valuable guidance that saw me pass my confirmation.

I would like to thank Janet Wiles who encouraged and supported me in applying for and receiving a UQ ResTeach scholarship, and Peter Lindsay who provided me with part-time employment at the ARC Centre for Complex Systems — these positions provided me with much needed financial stability during the first several years of the thesis.

I would like to gratefully thank Ian Hayes for taking me on and acting as my primary advisor through to submission and conferral — I very much appreciate the patience he has shown, and am thankful for the excellent and thorough feedback he has provided on numerous thesis drafts.

I would like to thank Peter Robinson and Ben Matthews for acting as my associate advisors and for providing an extremely useful third perspective when presented with thesis drafts to read. I would also like to thank Stephen Viller for acting as my thesis committee chair, and I would like to thank Michael Bulmer for advice regarding the included statistical results.

Finally, I would like to thank my parents for providing me with endless encouragement and support during my studies.
Keywords
visuo-spatial programming, programming environments, software navigation, spatial memory, software visualization

Australian and New Zealand Standard Research Classifications (ANZSRC)
ANZSRC code: 080309 Software Engineering, 30%
ANZSRC code: 080309 Programming Language, 35%
ANZSRC code: 080602 Computer-Human Interaction, 35%

Fields of Research (FoR) Classification
FoR code: 0803 Computer Software, 65%
FoR code: 0806 Information Systems, 35%
## Contents

1 Introduction ........................................... 19
   1.1 Problem statement ................................ 23
   1.2 Thesis ............................................. 23
   1.3 Empirical approach ................................ 26
      1.3.1 Hypothesis 1 – improved software navigation 26
      1.3.2 Hypothesis 2 – improved software structure 27
   1.4 Scope .............................................. 27
   1.5 Contributions ..................................... 28
   1.6 Thesis overview ................................... 29

I Background ............................................ 31

2 Human memory and mental models ..................... 32
   2.1 Human memory ..................................... 32
      2.1.1 Short-term memory ............................ 32
      2.1.2 Long-term memory ............................ 33
   2.2 Mental models ..................................... 34
      2.2.1 Cognitive maps ................................ 35
      2.2.2 Spatial modelling ............................... 36
      2.2.3 Narrative modelling ............................ 36
      2.2.4 Visual, simulation, and analogical modelling 36
      2.2.5 Human reasoning ................................ 37
   2.3 External representations of mental models ........ 37
   2.4 Summary ............................................ 38

3 Programmer cognition .................................. 39
   3.1 Programming as problem solving .................. 39
   3.2 Software mental models ............................ 40
   3.3 Mental imagery ..................................... 41
   3.4 Spatial memory and navigation .................... 42
   3.5 Code as an external representation .............. 43
   3.6 Summary ............................................ 43
# Visuo-spatial programming interfaces

## 7.1 Spatial memory

## 7.2 Requirements for visuo-spatial interfaces

- **7.2.1** Visual distinctiveness of programming artefacts
- **7.2.2** Spatial separation of programming artefacts
- **7.2.3** Ensuring a spatially consistent environment
- **7.2.4** Temporality
- **7.2.5** Programming landmarks

## 7.3 Design decisions for visuo-spatial interfaces

- **7.3.1** Unstructured or structured interface?
- **7.3.2** How to choose codebase subset?
- **7.3.3** How long is spatial consistency maintained?

## 7.4 Existing visuo-spatial programming environments

- **7.4.1** The text editor
- **7.4.2** Code Thumbnails
- **7.4.3** Code Canvas
- **7.4.4** Code Bubbles
- **7.4.5** Patchworks

## 7.5 Summary

# Method-flow visualisation

## 8.1 Comprehension of hyperspace structures

## 8.2 Current mitigation strategies

## 8.3 Flow views

## 8.4 The method-flow visualisation technique

- **8.4.1** Benefits during comprehension, navigation, and composition
- **8.4.2** Support for visuo-spatial programming
- **8.4.3** Implementation specific concerns

## 8.5 Summary

# The ‘Software Dimensions’ concept

## 9.1 Mapping to three ‘Software Dimensions’

- **9.1.1** The 1st dimension
- **9.1.2** The 2nd dimension – composition/methods
- **9.1.3** The 3rd dimension – inheritance

## 9.2 Supporting the ‘Software Dimensions’

- **9.2.1** Supporting the 1st dimension
- **9.2.2** Supporting the 2nd dimension
- **9.2.3** Supporting the 3rd dimension

---

10
12.1 Treatments .......................................................... 128
  12.1.1 Task choice .............................................. 128
12.2 Participants ...................................................... 128
12.3 Data collected .................................................... 129
12.4 Variables .......................................................... 129
  12.4.1 Implicit navigations .................................. 129
  12.4.2 Direct navigations .................................. 130
  12.4.3 Relative navigations .................................. 131
  12.4.4 Scrolling navigations .................................. 131
  12.4.5 Editing behaviour .................................... 132
  12.4.6 An example session transcript ...................... 132
12.5 Experimental design .......................................... 133
12.6 Analysis .......................................................... 134
12.7 Summary .......................................................... 134

13 An investigation of software navigation 135
13.1 Background ...................................................... 136
  13.1.1 Technologies under investigation .................. 136
  13.1.2 Alternative technologies ............................... 139
  13.1.3 Related studies ......................................... 139
13.2 Experiment planning .......................................... 141
  13.2.1 Goal ....................................................... 141
  13.2.2 Participants ............................................. 141
  13.2.3 Experimental equipment and materials ............ 141
  13.2.4 Tasks ..................................................... 142
  13.2.5 Solutions ................................................ 143
  13.2.6 Hypotheses .............................................. 144
  13.2.7 Experimental design .................................. 144
  13.2.8 Procedure ................................................ 145
  13.2.9 Data collection ......................................... 145
  13.2.10 Analysis procedure ................................... 146
13.3 Execution deviations ........................................... 146
13.4 Participant summary ........................................... 146
13.5 Quantitative analysis .......................................... 148
  13.5.1 Results by participant ............................... 148
  13.5.2 Results by task ........................................ 149
  13.5.3 Results by environment ................................ 149
  13.5.4 Results of the repeated measures t-tests .......... 150
  13.5.5 Summary of results .................................... 151
13.6 Qualitative analysis of Visuocode (only) ........................................ 151
13.7 Observations of disorientation ...................................................... 152
13.8 Post-study questionnaire ............................................................. 153
13.9 Findings ....................................................................................... 156
  13.9.1 Visuocode encouraged relative navigation ................................. 157
  13.9.2 Eclipse encouraged scrolling navigations ................................. 157
  13.9.3 Task affects navigation strategy ............................................. 157
13.10 Discussion ............................................................................... 157
  13.10.1 Threats to validity .............................................................. 159
  13.10.2 Lessons learned .................................................................... 160
13.11 Conclusions ........................................................................... 160

14 An investigation of program composition ........................................ 162
  14.1 Background ............................................................................. 163
  14.1.1 Technologies under investigation ......................................... 163
  14.1.2 Alternative technologies ..................................................... 164
  14.1.3 Related studies .................................................................... 164
  14.2 Experiment planning ............................................................... 164
  14.2.1 Goals .................................................................................. 164
  14.2.2 Participants ......................................................................... 164
  14.2.3 Experimental equipment and materials .................................. 165
  14.2.4 Tasks .................................................................................. 165
  14.2.5 Hypotheses ......................................................................... 165
  14.2.6 Experiment design ............................................................... 166
  14.2.7 Procedure ........................................................................... 166
  14.2.8 Data collection ..................................................................... 167
  14.2.9 Analysis procedure ............................................................. 167
  14.3 Execution deviations ................................................................. 167
  14.4 Participant summary ................................................................. 168
  14.5 Quantitative analysis ................................................................. 170
  14.5.1 Creation of code structure .................................................. 170
  14.5.2 Results by session ................................................................ 171
  14.5.3 Results by task .................................................................... 172
  14.5.4 Results by environment ........................................................ 173
  14.5.5 Results of the repeated measures t-tests ................................. 175
  14.5.6 Summary of results ............................................................. 176
  14.6 Qualitative analysis of Visuocode (only) .................................... 176
  14.7 Post-study questionnaire ........................................................... 177
  14.8 Findings .................................................................................. 179
14.8.1 More software structure created while using Eclipse .......................... 179
14.8.2 More relative navigations while using Visuocode .............................. 179
14.8.3 More direct navigations while using Eclipse .............................. 180
14.8.4 More scrolling navigations while using Eclipse .............................. 180
14.8.5 Navigation behaviour was affected by task .............................. 180

14.9 Discussion ............................................................................ 180
14.9.1 Threats to validity .......................................................... 182
14.9.2 Lessons learned ............................................................. 183

14.10 Conclusions ................................................................. 183

V Conclusion ............................................................................ 185

15 Discussion ............................................................................. 186
15.1 Related research ................................................................ 186
15.2 Theory .............................................................................. 187
15.3 Implementations ............................................................... 188
15.4 Formal evaluation ............................................................. 189
15.5 Good versus bad navigations .................................................. 190
15.6 Reflection on empirical approach ............................................ 191
15.6.1 Benefits ...................................................................... 191
15.6.2 Limitations .................................................................. 192
15.7 Limitations of Visuocode .......................................................... 193
15.7.1 Visuocode does not support the 3rd Software Dimension .................. 193
15.7.2 Source code lag ............................................................ 194
15.7.3 No support for reverse navigation ......................................... 194
15.7.4 One flow view .............................................................. 195
15.7.5 No support for search ..................................................... 195
15.8 Implications ...................................................................... 196
15.8.1 Increased support for relative navigation ................................. 196
15.8.2 Better representation for the composition hierarchy ..................... 196
15.8.3 Enhanced representation of class outlines .................................... 196

16 Conclusions .......................................................................... 197
16.1 Impact ............................................................................... 198

17 Future work .......................................................................... 199

Appendices .............................................................................. 199

A Study task sheets .................................................................... 200
B Study 1 participant summaries
   B.1 Navigation summaries .............................................. 205
   B.2 Participant use of method-flow .................................... 212

C Study 2 participant summaries
   C.1 Navigation summaries .............................................. 219
   C.2 Participant use of method-flow .................................... 229

List of Figures

6.1 A C++ program that sums integer arguments ....................... 69
6.2 A C program that calculates the summed area of a rectangle and circle .... 70
6.3 A Java program that calculates the summed area of a rectangle and circle .... 71
6.4 For illustrative purposes: the Visuocode class composition model .......... 72
6.5 An Eclipse IDE window with source files manually arranged ............ 74
6.6 Screenshots comparing Eclipse on an 11 inch laptop to a 27 inch display .... 74
7.1 Code Thumbnails [DCM+06] ............................................ 87
7.2 Code Canvas [DR10] .................................................. 89
7.3 Code Bubbles [BRZ+10a] ............................................. 90
7.4 Patchworks [HF14] .................................................. 91
8.1 A flow view is a horizontally scrollable view containing adjacent view columns . 94
9.1 The three ‘Software Dimensions’ ..................................... 99
9.2 Supporting the 2nd Software Dimension ................................ 103
9.3 Supporting the 3rd Software Dimension ................................ 104
9.4 A hypothetical cognitive map ......................................... 105
10.1 A Code-flow window with three source columns in a flow view ........ 109
11.1 Visuocode consists of two types of window .......................... 113
11.2 Visuocode Flow window ............................................... 114
13.1 The Visuocode Workspace Manager and Flow windows ............... 137
13.2 The Eclipse IDE .................................................... 138
List of Tables

1.1 An outline of the chapters contained within each part of the thesis 30
5.1 A comparison of features provided by a representative set of environments 63
12.1 Symbols representing navigation behaviour codes 130
12.2 An example session transcript 132
12.3 An example of the navigations performed by a participant 132
12.4 An example summary of the navigations performed by a participant 133
13.1 Study 1: Size of task source code 142
13.2 Study 1: Description of activities 143
13.3 Study 1: Experiment design 145
13.4 Study 1: Navigation strategies used by participants 146
13.5 Study 1: Maximum navigation depths 147
13.6 Study 1: Summary of navigations grouped by participant 148
13.7 Study 1: Navigations grouped by participant 149
13.8 Study 1: Summary of navigations performed during each task 149
13.9 Study 1: Navigations performed during each task 149
13.10 Study 1: Summary of navigations performed using each environment 150
13.11 Study 1: Navigations performed using each environment 150
13.12 Study 1: Summary of results from repeated measures t-test 151
13.13 Study 1: Results of the System Usability Scale 154
13.14 Study 1: Results of the post-study questionnaire 155
14.1 Study 2: Experiment design 166
14.2 Study 2: Phase durations for each participant 168
14.3 Study 2: Number of classes and methods created by each participant 171
14.4 Study 2: Summary of navigations performed during each session 171
14.5 Study 2: Summary of navigations performed per minute during each session 172
14.6 Study 2: Average of navigations performed per minute during each treatment 172
14.7 Study 2: Summed averages of navigations performed per minute during each task 173
14.8 Study 2: Navigations performed during each task 173
14.9 Study 2: Summed averages of navigations performed using each environment 174
14.10 Study 2: Navigations performed while using each environment 174
Glossary

**AST** Abstract Syntax Tree. 18

**CSD** Control Structure Diagram. 18

**CSV** Comma Separated Value. 18

**HCI** Human-Computer Interaction. 18

**IDE** Integrated Development Environment. 18, 54, 68

**MDI** Multiple Document Interface. 18
Chapter 1

Introduction

Software is increasingly crucial to modern civilisation because every industry – from health to transportation – is now reliant on the correct functioning of software-controlled computer systems. The importance of software has long been appreciated. During the late 1960s, there was a realisation that the nascent software industry was facing a crisis as the computer industry had neither the experience nor expertise needed to develop the increasingly complex software that was being asked of it. This realisation brought about the first NATO Software Engineering Conference, which spurred attempts to make software development an engineering discipline [NR68].

The term software engineering [NR68] describes an ideal of developing software using an engineering approach with a staged process that includes requirements gathering, software specification, high- and low-level design, software implementation, acceptance testing, and deployment. It was hoped that such an approach would allow software projects to be carried out within time and budget.

While software engineering techniques have led to dramatic improvements in the initial development of software, the cost of ongoing maintenance has remained significant and disproportionate compared to the cost of initial development [BDKZ93]. It is thought that this is due to a process referred to as software ageing, which is characterised by the gradual deterioration of software quality [Par94]. It seems that either software is of poor quality when completed – making it difficult to modify; or the quality of the software degrades as modifications occur during maintenance. This degradation is reflected in increased software complexity, which leads to reduced clarity during future maintenance.

Software ageing is hard to mitigate as there is no agreed way of objectively measuring software quality. Due to the expense of early computer systems, the earliest concept of software quality related to efficiency – measured by the amount of computer memory used and the duration of program execution. This concept of quality resulted in programmers producing overly complicated code in order to make their programs as efficient as possible [Dij68a, Wei74]. As software requirements increased, and computer hardware became cheaper, it became increasingly apparent that this mindset was limiting the achievable size of software projects [Dij72].
While efficiency remains important, it is now realised that a more important quality of software relates to its maintainability – in part because of the difficulty of ensuring that software specifications meet actual requirements, which often leads to changes. Although there are many anecdotal recommendations for developing high quality software [McC04, Mar09], such recommendations are generally special cases of the following three principles: software should be decomposed in a modular fashion to allow a division of effort [Par72a], be structured hierarchically to avoid circular dependencies [Dij68b], and use appropriate abstractions to facilitate information hiding [Par72b, Par02]. In short, the quality of software is directly related to its structure.

Unlike real world objects that have a physical shape and structure, software has a file-based structure that reflects how its source code files are stored in the directory hierarchy, as well as an emergent structure that results from the semantic relationships between code definitions as they are added to the software – I use the term ‘emergent’ because this structure emerges as a codebase is modified. Most programming languages store source code within plain-text source code files that define functions and methods that are invokable from one another. A program’s logical network structure of invocations is referred to as its call graph. Source code files may also define data structures that include or inherit from data structures declared in other source code files – such composition and inheritance create two other logical tree structures referred to as the composition hierarchy, and the inheritance hierarchy, respectively. Another aspect of software structure is that objects can be passed down the call graph as arguments to method calls, which may then have method calls invoked upon them. Such data-flow characteristics can conceptually lead to chunks of software structure being replicated in different areas of a piece of software’s emergent structure. Additionally, local, class, and global variables may be referenced from within methods by either invoking methods or directly accessing members. These different aspects of the emergent structure of software make it a multi-dimensional artefact similar to hypertext documents – such a structure is often referred to as a hyperspace [EH89, KH95, JB07]. It is emphasised that the emergent structure of software does not equate to the call graph. However, depending on the dominant programming paradigm used for a program, the call graph may reflect its emergent structure more closely than for others. For example, the call graph of a program written in a procedural fashion reflects its emergent structure to a greater degree than if that program were written in an object-oriented manner that makes extensive use of inheritance and class composition.

During software development, programmers often want to move between source code definitions, for example: following a method invocation to a method definition in another source code file, navigating to the class that the current class inherits from, investigating the definition of a member variable, or finding, and navigating to, a method that calls the current method. Each such action is referred to as a software navigation, and often necessitates opening a different source code file. This thesis distinguishes between several different modes of navigation that are used to navigate software structure: direct navigation occurs if a programmer navigates
directly to a declaration using an auxiliary view, relative navigation occurs if the environment allows a programmer to navigate according to the emergent structure of the software, scrolling navigation occurs if a programmer scrolls through a file between method definitions or other declarations, and implicit navigation occurs if a programmer moves their attention between declarations, definitions, or windows, with no explicit user input. It is emphasised that for the purposes of this thesis, navigations are categorised based on how the programmer performed the navigation not why the programmer performed the navigation. For example, it is considered a direct navigation if a programmer uses an auxiliary view to navigate to the definition of a called method because even though the programmer desires to navigate relatively along the call graph they do so by performing a direct navigation. Similarly, if a programmer desires to consult the declaration of an object member, and scrolls an editor to do so, it is considered a scrolling navigation.

In most software development environments, direct navigations are often achieved by using an auxiliary tree representation of the software file structure, referred to as a file browser – an example is the Eclipse Package Explorer. Navigations performed using other auxiliary views, including searches, are also considered direct navigations. At the start of a programming session, a direct navigation is usually performed to navigate to an initial source code file, and possibly a specific location within it. Relative navigations are performed to traverse the emergent structure of software – such as traversing the call graph by navigating to a called method, or navigating to the class definition of a variable. For example, while using Eclipse, if an OS specific modifier key is held down while the mouse is hovered over a method call, the method call is turned into a hyperlink that opens the corresponding source code file in a new editor tab. However, due to object polymorphism in object-oriented languages, it may not always be possible for an environment to determine which method definition corresponds to the method executed – such as if calling a method via an interface, or calling a virtual method. In such cases, the programmer must determine which source code file corresponds to the reference, and perform a direct navigation to it. Scrolling navigations are performed if the programmer scrolls a code editor between declarations – for example, between methods, or to the top of the file to refer to a class member. Implicit navigations occur if a programmer shifts their focus without the need for an explicit user input – for example, between two open windows, or between methods that are located near enough to one another they can be viewed without scrolling the code editor.

Several studies [KAM05, K+11, KKK+13] have revealed that programmers often use a two-phase strategy while exploring new code. First they carry out an exploration phase during which they opportunistically investigate source code files until they find a section of code that is potentially relevant – referred to as an anchor point. Next they begin a traversal phase during which they explore the call graph from the anchor point. Programmers may perform many of these explorations and traversals while building up a working set of source code files relevant to the current task [RCM04].
During software navigation, programmers may become disoriented by either becoming lost in the source code or losing task awareness [dAM06]. Similar problems have been observed of users navigating hypertext systems [Con87]. It is thought that navigational disorientation is caused by trying to navigate a large information space using the medium of a narrow computer display [Ho02] – this is referred to as the keyhole phenomenon [Ho02, DAM05] – and is exacerbated if an environment also allows navigation of irrelevant system source code [BC92]. Another cause of disorientation is if there is little visual spatial continuity when navigating between individual documents or source code files – this is referred to as a lack of visual momentum [Ho02, DAM05].

De Alwis [DAM05] distinguishes between these two types of disorientation: when caused by user interface problems such as a lack of visual momentum, it is “better characterised as a lose of situational awareness, as they relate to comprehending changes in the external world” [DAM05, p. 9–10]; in contrast, when caused by the keyhole phenomenon, it “results from difficulties in the mental world, a difficulty in situating one’s self in context” [DAM05, p. 10].

This suggests that there are two problems that need to be addressed: firstly, visual momentum needs to be better supported in development environments so that programmers are able to maintain situational awareness; secondly, spatial memory needs to be better supported so that programmers are less likely to become lost. Existing suggestions to mitigate these problems include: improving transitions so that navigations are more continuous [Ho02], and the provision of an auxiliary view that provides a map of nodes that are navigated between [Con87] – such as Prodet [AFQ+15].

In order to better understand the causes of disorientation while navigating hypertext documents, Edwards & Hardman [EH89] had 27 participants navigate through one of three variants of a set of hypertext documents – one set featured hierarchical linking of subject matter, another required users to navigate via an index page to access other topics, and the third (mixed condition) provided both sets of links. They found that the participants who use the mixed condition reported more instances of disorientation than those who used either of the other two conditions. They concluded that the individuals appeared to be “attempting to create cognitive representations of the hypertext in the form of a survey-type map” [EH89, p. 104], and that having multiple types of link, i.e., hierarchical and index, negatively affected the formation of such a cognitive map. They recommended that “readers should be allowed to develop a cognitive map of one view of the data structure before being given the option of navigating through the data some other way” [EH89, p. 104]. Due to the fact that software source code forms a hyperspace data structure similar to hypertext, these findings suggest that software development environments should emphasise a primary navigation structure to better support the formation of an initial cognitive map during software exploration.

There has been extensive research into the cognitive processes of programmers during program comprehension and composition, however, there has been little investigation of the different types of mental models used during programming. While it is generally acknowledged that source code files lack a natural spatial distribution that would allow a programmer to
navigate between them in the normal sense, it has been recognised that source code files do contain spatial characteristics, such as landmarks in the form of comments, that may be used by programmers to aid navigation [CFO05]. There is also growing evidence that spatial ability correlates with potential programming ability [FCZ06, TBH06, JB07], which supports the belief that a programmer may use a spatial cognitive map during software navigation.

In mainstream development environments, the code editor is the primary tool that is used by programmers to view software source code, and to create new software structure by defining new classes, methods, and functions within source code files. Despite its importance, and having been described as one-dimensional [Mye86], the way that source code is represented by the editor functionality of mainstream development environments for text-based languages has changed little since text editors were originally developed – while using such environments, navigating to another source file often causes the existing programming context – the code editor containing the current source file – to be obscured. This lack of visual momentum makes it difficult for programmers to remain oriented in the source code. Even though features such as tabs and bookmarks allow programmers to form a working set of source code files relevant to the current task, they do not support the programmer in forming a cognitive map of the software. It is proposed that the reverse may also be true during program composition – due to being unable to form a consistent spatial cognitive map, software structure may be compromised as it is constructed.

1.1 Problem statement

While navigating software, programmers may become lost and disoriented due to development environments having low visual momentum during navigation transitions, and the size and irregularity of the emergent structure of software. Two key problems lead to this disorientation:

1. The visual momentum problem – mainstream development environments do not provide enough visual continuity when transitioning between source code files; and
2. The hyperspace problem – the irregular structure of software makes it difficult to integrate software navigations into a consistent cognitive map within long-term memory.

This thesis investigates how to provide better support for the integration of software navigations into a spatial cognitive map within long-term memory to better support software navigation.

1.2 Thesis

In Dijkstra’s paper, “The Humble Programmer” [Dij72] he argues that it is only through abstraction that we can build software that would otherwise be too complex for human abilities.
He writes:

“...the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.” [Dij72, p. 864]

and:

“Up until now I have not mentioned the word ‘hierarchy’, but I think that it is fair to say that this is a key concept for all systems embodying a nicely factored solution. I could even go one step further and make an article of faith out of it, viz, that the only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution.

“Hierarchical systems seem to have the property that something considered as an undivided entity on one level is considered as a composite object on the next lower level of greater detail; as a result the natural grain of space or time that is applicable at each level decreases by an order of magnitude when we shift our attention from one level to the next lower one.” [Dij72, p. 865]

When applied to modern object-oriented programming languages, the hierarchy that Dijkstra is describing is the composition hierarchy that is formed by classes incorporating other classes as members. As stated by Dijkstra, it is necessary that software be nicely factored, however, appropriate factoring of software is easiest during the design and implementation phases of software development while developers have a whole system perspective that may be supported by up-to-date documentation. In contrast, maintenance programmers often do not have such a whole system perspective, and must rely on their development tools to understand how a system is structured because documentation is often not adequately updated. However, despite the importance of software structure, mainstream development environments do not sufficiently support its visualisation, navigation, comprehension, or creation. Without such support, how can maintenance programmers be expected to maintain a nicely factored composition hierarchy?

The thesis of this project is that the call graph and composition hierarchy of software provide a consistent and canonical structure that can be used to confer a spatial relationship between source code declarations, which, when adequately visualised, can better leverage visuo-spatial working memory and allow better utilisation of long-term spatial memory, facilitating software creation, navigation, comprehension, and modification. The aim of this project is to develop a prototype software development environment that can visualise the composition hierarchy of software while also providing increased support for visual momentum and mitigating the effects of the hyperspace problem. Such an environment should provide better support for software navigation and composition, improving programmer productivity.
Specifically, it is believed that mainstream development environments suffer low visual momentum because they do not effectively leverage visuo-spatial memory – the visuo-spatial sketchpad is the name given by Baddeley and Hitch to the theoretical component of working memory that is able to keep track of both the location and appearance of objects if they are no longer visible [Bad83, Bad12]. It is proposed that a way to better leverage visuo-spatial memory is to allow code editors to be presented side-by-side within a horizontally scrollable view, thereby increasing visual momentum. The benefit is that a programmer is able to explore the emergent structure of the software while viewing the source code of related declarations side-by-side, make any required modifications, then return to their initial programming context.

During program composition, such a view would also allow the creation of new software structure within an adjacent editor. If editors are scrolled off the side of the view, as the spatial position of each editor relative to one another remains consistent, short-term visuo-spatial memory in conjunction with long-term spatial memory will be able to keep track of the editors even though they may no longer be visible.

During this project, the method-flow visualisation technique has been developed, and implemented within the Code-flow software exploration tool, and the Visuocode prototype development environment [BH13]. Method-flow allows a programmer to navigate the composition hierarchy or call graph of a program using adjacent editor columns that are contained within a horizontally scrollable flow view. Object references and method calls are represented as hyperlinks that, if followed, cause the corresponding class or method to be displayed in a new editor column adjacent to the original programming context. The programmer may then scroll the flow view back and forth to see previously visited editor columns containing classes and methods. Thus, method-flow allows the programmer to navigate software based on its emergent structure instead of its file structure, while also better supporting spatial memory by maintaining a spatially consistent trail of any methods traversed. Method-flow has been developed to address the visual momentum problem.

Although increased visual momentum should provide better support for visuo-spatial memory which, in turn, should support the integration of individual software navigation journeys into a spatial cognitive map, the hyperspace problem may hinder the development of such a map. The Software Dimensions concept is also introduced, which imposes a spatially regular structure on the naturally irregular hyperspace structure of software. The Software Dimensions are intended to inform the design and development of a software development environment that implements the method-flow visualisation technique in order to optimise the integration of relative navigations into a consistent cognitive map. The Software Dimensions concept has been developed to address the hyperspace problem.

Visuocode is considered to be just one example of a style of development environment that explicitly supports visuo-spatial memory. The term visuo-spatial programming is introduced to refer to the use of such an environment that leverages both short-term visuo-spatial memory and long-term spatial memory.
In summary, a visuo-spatial programming interface refers to a user interface that ensures spatial consistency in order to better leverage spatial memory. Method-flow is a software visualisation technique that increases the visual momentum of the programming interface to optimise the support for short-term visuo-spatial memory facilitating integration of software navigations into long-term spatial memory. The Software Dimensions are a way of conferring a spatial structure to software, based on its emergent structure, so that it is more amenable to integration into a consistent cognitive map. These concepts have informed the design and guided the development of the Code-flow software exploration tool and the Visuocode prototype software development environment whose aim is to support software navigation in order to improve programmer productivity.

1.3 Empirical approach

It is emphasised that the aim of this project is provide better support for software navigation. Therefore the empirical evaluations of Visuocode are focused on identifying differences in programmer behaviour while participants are completing software navigation and program composition tasks.

Understanding how Visuocode affects the cognitive map of a programmer is not an aim of this project due to the obvious challenges involved with analysing a programmer’s cognitive state. While it is assumed that participants show fewer signs of disorientation if they are able to form a consistent cognitive map, and conversely more signs of disorientation if not, reports from previous studies investigating disorientation have warned that disorientation can be hard to identify through observation, and that self-reporting of disorientation by participants may affect their behaviour [DAM05]. Therefore the studies described in this dissertation are classed as exploratory studies, and programmer behaviour is characterised based on a quantitative analysis of the number and kind of navigation participants performed, as well as a thorough qualitative analysis of participant sessions.

This thesis describes two exploratory think-aloud studies that investigate differences in programmer behaviour. The first study investigates software navigation motivated by software understanding tasks, and the second investigates programmer behaviour during program composition tasks. The aim of these studies is to develop an initial characterisation of the difference in navigation behaviour while using a traditional mainstream environment (Eclipse) versus a prototype development environment that provides explicit support for relative navigation (Visuocode). Limitations of this approach are discussed further in Section 15.6.

1.3.1 Hypothesis 1 – improved software navigation

The first hypothesis of this thesis is that while using method-flow for software understanding, programmers will perform more relative navigations than otherwise due to method-flow mitigat-
ing disorientation by providing explicit support for remaining oriented during relative software navigation. During maintenance activities, programmers often navigate down a program’s call graph to find how, or if, functionality is implemented. Once their task is completed, they often become disoriented while attempting to return to a previously visited section of code, as they cannot remember how they got to the source file in the current editor. The programmer then resorts to performing a direct navigation to a known location before navigating to the location they originally intended. It is thought that this disorientation occurs because traditional environments do not adequately remind the programmer of the methods that they have traversed [KKK+13]. To compensate, programmers favour direct navigations as they require the programmer to lay a mental trail of explicitly labelled navigations – i.e., the programmer needs to explicitly search for the method they desire by first locating the class using its name, then the method using its name, which results in those names being rehearsed into long-term memory as a route [DeL05]. While using method-flow, programmers should not need to navigate in this manner because they can easily scroll the flow view to refresh their memory of the methods they have traversed, and therefore easily backtrack to follow a different call graph branch.

1.3.2 Hypothesis 2 – improved software structure

The second hypothesis of this thesis is that, if using method-flow for program composition, programmers will be less likely to inappropriately incorporate code into the current programming context. During program composition, it is suspected that programmers often create methods that are overly long, or complex, to avoid the mental and navigational overhead associated with creating a new method. Similarly, to avoid the overhead of creating a new class, it is suspected that programmers often implement code within a new method that should be extracted out into a new class. While using method-flow, programmers should be less likely to include inappropriate code within the current class because a new class or method can be easily created juxtaposed adjacent to the existing programming context. If method-flow does provide useful support for creating new software structure, and reduces the tendency for programmers to create inappropriately large methods, it is expected that programmers should instead create more, but smaller, methods than otherwise. Therefore, the number of methods and classes created during program composition may be considered a proxy measurement of software structure quality. It is emphasised, however, that this claim is only expected to hold for small programs because as programs become larger better software structure could reduce the number of source files needed.

1.4 Scope

This thesis focuses on the navigation, comprehension, and composition of object-oriented software that might be written in the Java programming language. This thesis does not consider
navigation through software systems that act in concert, such as web systems where relevant code may execute within a browser as Javascript, within a web server in Java, and within a database as an SQL Stored Procedure or query. Additionally, this thesis focuses on the visualisation of the static structure of software, and therefore does not consider dynamic aspects such as the use of callbacks or other concurrency primitives; nor does it consider threading aspects such as understanding the state interactions of two different threads, or the related issue of locking.

1.5 Contributions

This project has yielded research contributions in the form of theory, implementations, and empirical results.

1. Requirements for *visuo-spatial programming interfaces* have been detailed that ensure that such interfaces properly support visuo-spatial memory.

2. The method-flow visualisation technique has been developed to mitigate the visual momentum problem by allowing a programmer to traverse a call graph using adjacent editor columns within a scrollable flow view.

3. The Software Dimensions concept has been developed to mitigate the hyperspace problem by conferring a consistent spatial structure to software by refining the concept of *layers of abstraction* to take account for the difference between abstraction through composition versus inheritance.

4. Two software development tools implementing method-flow have been produced: Codeflow, a software exploration tool; and Visuocode, a prototype software development environment.

5. An *experiment plan* including a qualitative protocol and analysis technique has been developed for investigating navigation behaviour during programming tasks.

6. Two qualitative studies of programming have been carried out. The first study investigated software navigation behaviour exhibited by participants attempting software understanding tasks – it was found that participants favoured relative navigations while using Visuocode, and favoured direct and scrolling navigations while using Eclipse. The second study investigated software navigation and creation behaviour exhibited by participants attempting program composition tasks – unexpectedly, participants created less software structure while using Visuocode (not more), however, it is suspected this result was confounded by too small a task size. The second study also repeated the findings of the first study regarding navigation behaviour.
1.6 Thesis overview

For ease of navigation, this thesis has been separated into several parts – each of which includes two or more chapters. Part I (Background) provides a summary of relevant background literature including an overview of human memory and mental models, programmer cognition, software visualisation, and software development environments. Part II (Theory) describes the theory that was developed during this project including a description of visuo-spatial programming interfaces, the method-flow visualisation technique, and the Software Dimensions concept. Part III (Implementations) describes the Code-flow and Visuocode software development tools that implement method-flow, which were developed during this project. Part IV (Formal Evaluation) describes the experiment plan that was developed to evaluate Visuocode, as well as two qualitative studies that investigate programmer behaviour – the first of software navigation, and the second of program composition. Part V (Conclusion) first provides a discussion of important concepts within the thesis including a review of the empirical results, then presents the conclusions reached, and proposes future work. Table 1.1 shows the titles of each chapter of the thesis.
| Part I – Background | Chapter 2 – Human memory and mental models  
|                     | Chapter 3 – Programmer cognition  
|                     | Chapter 4 – Software visualisation  
|                     | Chapter 5 – Software development environments  
| Part II – Theory   | Chapter 6 – About programming  
|                    | Chapter 7 – Visuo-spatial programming interfaces  
|                    | Chapter 8 – Method-flow visualisation  
|                    | Chapter 9 – The ‘Software Dimensions’ concept  
| Part III – Implementations | Chapter 10 – Code-flow  
|                       | Chapter 11 – Visuocode  
| Part IV – Formal evaluation | Chapter 12 – A qualitative experiment plan  
|                       | Chapter 13 – An investigation of software navigation  
|                       | Chapter 14 – An investigation of program composition  
| Part V – Conclusion | Chapter 15 – Discussion  
|                    | Chapter 16 – Conclusions  
|                    | Chapter 17 – Future work  
| Appendices         | Appendix A – Study task sheets  
|                    | Appendix B – Study 1 participant summaries  
|                    | Appendix C – Study 2 participant summaries  

Table 1.1: An outline of the chapters contained within each part of the thesis
Part I

Background

The aim of this thesis was to develop a software development environment that provides additional support for the visualisation, navigation, comprehension, and creation of code structure by better leveraging visuo-spatial memory. Therefore it has been necessary to not only review similar tools that have been developed within computer science, but also to review previous research into the psychology of programming, as well as research in cognitive science related to human memory and cognition.

The four chapters in this part are written so as to not contribute any new theory, however, if the reader does choose to skip any of these chapters they are advised to read each chapter summary to ensure that they are aware of any pertinent points that have been highlighted. Chapter 2 reviews research into human memory and mental models. Chapter 3 reviews research into programming from the cognitive psychological perspective. Chapter 4 reviews the field of software visualisation. Chapter 5 reviews software development environments.
Chapter 2

Human memory and mental models

While a comprehensive review of research into human memory and mental models is beyond the scope of this report, this chapter does review material that is considered relevant to programmer cognition. Programming is a cognitive activity that must be performed within the constraints of working memory, and that produces as its output an external representation in the form of source code. However, unlike many problem solving tasks, programming also requires access to extensive knowledge from long-term memory including domain knowledge related to the programming problem, as well as programming knowledge such as programming language syntax and semantics.

This chapter reviews aspects of cognition that are relevant to programming. Section 2.1 reviews the current understanding of human memory. Section 2.2 reviews mental models. Section 2.3 reviews research related to external representations of mental models.

2.1 Human memory

Traditionally, two main types of human memory have been recognised: short-term memory (STM), which is volatile and limited to a small number of items; and long-term memory (LTM), which is thought to be effectively permanent [Mil56]. This dichotomy is supported by neuropsychological patients who present differing short-term and long-term memory impairments [SM57].

2.1.1 Short-term memory

Several models have been developed that fractionate short-term memory into sub-components that are able to process information autonomously [AS68, BH74, Log11]. Baddeley and Hitch’s model [BH74], now referred to as the Multiple-component model of Working Memory (M-WM) [Bad03], contains several sub-components that have been identified through investigation of patient cases, and confirmed through formal empirical studies of healthy participants. Early work identified the phonological loop – which is able to store about seven phonemic (sound-
based) syllables that may be maintained through concentrated rehearsal [Mil56, BH74, Bad83, Bad03, Bad12], and the visuo-spatial sketchpad – which is able to temporarily store image impressions and spatial locations [Cor72, BH74, Bad83, Bad03, Bad12]. These systems appear to operate semi-independently of consciousness as, for example, visually presented written material may be automatically translated and entered into the phonological loop in phonemic form [Log11]. Similarly, visually presented material is cognitively grouped together based on the Gestalt principles of perceptual organisation [Tve05]. The third component is the Central Executive, which Baddeley describes as a ‘homunculus’ (or little man that makes the decisions) – it is essentially a catch-all for any cognitive ability that is not yet explained by identified components [Bad03].

Another component was later added to the multi-component model to explain the cognitive ability to ‘bind’ perceived visual, spatial, and auditory information into a single perception (such as the roar of a lion as it jumps at you), and to accommodate the recency effect [BH74] – the proposed episodic buffer binds information from the other components of working memory, as well as from long-term memory, into perceptual objects that may be consciously manipulated [Bad00]. It is now thought that it is the episodic buffer that provides the ability to increase the apparent capacity of short-term memory by ‘chunking’ information [Mil56]. The episodic buffer seems to have a limit of about four such composite objects [Bad12].

An issue with current models of working memory is that they do not comprehensively address how information is retrieved from long-term memory to be manipulated within working memory. Recently, it has been suggested that the phonological loop and the visuo-spatial sketchpad may actually operate independently of conscious control by initiating retrieval of information from long-term memory that is then made available to consciousness from the episodic buffer [Log11, Bad12]. There are (at least) two opposing views regarding how information from long-term memory is accessed consciously. Baddeley suggests that information from long-term memory is copied into working memory – he argues that this is why such information is able to be consciously manipulated, and also explains why patients with deficits of long-term memory are able to perform as well on some problem solving tasks as healthy individuals [Bad00, Bad12]. In contrast, others suggest that stimulus in working memory activates areas of long-term memory, which makes it available to consciousness [CE93, EK95].

### 2.1.2 Long-term memory

Long-term memory stores at least three different types of information: spatial information, such as representations of real world environments stored as cognitive maps [Tol48]; semantic information, such as language structure and vocabulary, as well as other information that is considered ‘knowledge’; and episodic memory, which stores an individual’s experiences [Tul72]. Neurologically, long-term memory is thought to be associated with the hippocampus – semantic and episodic information has been associated with the left hippocampus, while spatial memory
has been associated with the right [Mil71, BMO02].

Historically, each type of information stored in long-term memory has been investigated using differing methods. Research into spatial memory has tended to observe the behaviour of animals or participants within a variety of environments – for example, early work studied rats’ navigations through various enclosures [Tol48], while recent work has observed how children navigate novel environments [PLD+14]. Such research theorises what information the subject has available to them that leads them to make the navigation choices that they do. In contrast, research into semantic and episodic memory usually relies on an experimenter conversing with a study participant, or the participant’s abilities being measured in another way, such as via an interactive computer program.

While episodic and spatial memory are thought to be physically associated with different regions of the hippocampus, this does not necessitate that they are functionally disjoint. It has long been conjectured that spatial memory and episodic memory may have the same functional basis [Nei87, BMO02]. It might be said that spatial memory provides a map that allows the exploration of episodic memory as, if we think of a place, we can quickly remember experiences had there. Similarly, non-experiential information may be associated with spatial locations. Popular memory techniques, such as ‘the method of loci’, may be used to associate information with specific places, which may later be recalled by mentally returning to those places [Nei87].

### 2.2 Mental models

The term *mental model* has been used since at least the year 1900 [McC00] in a variety of contexts, to varying levels of formality, to refer to an individual’s understanding of a particular subject. In the context of spatial memory, mental models have been described as small-scale models of external reality [Cra43]. The term is now used to describe a number of different types of mental state used for differing cognitive processes that may access either short-term memory or long-term memory, or both [Ner02].

Mental models exist on a continuum from a direct mapping of physical reality to much more abstract models related to problem solving. Nancy Nersessian describes a mental model as:

“...an analogue in that it preserves constraints inherent in what is represented. Mental models are not mental images, although in some instances an accompanying image might be employed. The representation is intended to be isomorphic to dimensions of the real world system salient to the reasoning process. Thus, for example, in reasoning about a spring the mental model need not capture the three-dimensionality of a spring if that is not taken to be relevant to the specific problem-solving task.” [Ner02, p. 141]

Nersessian gathers together a small vocabulary that may be used to describe mental models. A mental model may be described as ‘iconic’ if it has a diagrammatic or imagistic form that is
an analogue to something that may potentially be seen or drawn. In contrast, a mental model is said to be ‘propositional’ if it relies on “a language like encoding possessing a vocabulary, grammar, and semantics” [Ner02, p. 142]. The elements of a mental model are described as ‘modal’ if they are “analogues of the perceptual states from which they are extracted” – for example, the image of cat [Ner02, p. 142]; or ‘amodal’ if they are “arbitrary transductions from perceptual states” – for example, the word ‘cat’ [Ner02, p. 142].

The term ‘mental model’ is also used to refer to structured semantic knowledge. According to Gentner et al.:

“A typical piece of mental model research is characterised by careful examination of the way people understand some domain of knowledge.” [GS14, p. 1]

Gentner’s definition relates to how knowledge is stored within long-term memory – perhaps as a hierarchy of associated concepts within semantic memory. This usage is closely related to how mental models are referred to in the context of program comprehension research, which is discussed in the next chapter. In contrast, the following sections describe several types of mental model that may be viewed as spanning both long-term memory and working memory, i.e., they relate to the use of information stored in long-term memory by cognitive processes manipulating working memory.

2.2.1 Cognitive maps

Proposed by Tolman, a cognitive map describes the internal mental representation of an external environment that is used for navigation purposes [Tol48]. Spatial information can be described as being either egocentric – relative to the person; or allocentric – relative to the environment [BMO02]. It is thought that during exploration of an environment, each journey results in egocentric information being processed into a ‘narrow strip map’, which is then integrated into an allocentric model of the environment, referred to as a cognitive map – Tolman suggests that failure of integration may be caused by a number of factors including brain damage [Tol48].

Such spatial memory is thought to be so highly developed because it has been crucial for the survival of higher order animals by allowing them to avoid predators, and remember where resources are (such as food and water). A key aspect of cognitive maps is that they are an internal representation of the real world. Therefore the coarse structure of the map only needs to be updated very rarely, while finer details, such as whether or not there is water in a particular stream, need to be updated more frequently. It is possible that the ability to update spatial memory with temporally relevant information may be the basis for episodic memory, while the ability to associate arbitrary information with such spatial locations may provide the capability for more abstract mental modelling.

An implicit prerequisite for spatial memory is that objects remain in a consistent position for some period of time. It stands to reason that permanent objects, through re-enforcement,
are stored as more permanent memories than objects that are moved. For example, buildings and roads remain in the same place for decades, while a car may be parked in a specific location for only a number of hours. Importantly, while a landmark may move, spatial memory relative to that landmark may remain consistent. For example, a person’s memory of the floor plan of a large ocean liner remains consistent even though the ship, itself, is moved to different ports. While spatial memory is a component of long-term memory, it is believed that during cognition ‘indices’ into spatial memory are manipulated within the visuo-spatial sketchpad [Bad00].

2.2.2 Spatial modelling

While Neisser thought of cognitive maps as providing “a sense of where you are” [Nei87, p. 195], humans are also able to mentally imagine themselves in other places, and also form map-like representations in their mind. However, while reasoning using such mental maps, people often make mistakes that indicate their spatial memory is stored in a hierarchical structure – for example, thinking that Reno is east of San Diego because generally Nevada is east of California. This has led to the theory that spatial memories at different scales are arranged in a hierarchical mental structure where each level represents a memorable landmark [MHH89].

2.2.3 Narrative modelling

While reading a story, a reader often builds up a cognitive map of the locations and places that are being described. Researchers have investigated how narrative can be used to build such artificial cognitive maps by describing a scene to a participant, then testing their memory by asking them questions about the scene. Often, such a narrative places the individual within an environment, such as a theatre, and describes objects that are placed around them; the participant is then asked questions about the scene before and after being asked to imagine themselves moving about, as well as changing their orientation [FT90]. Narrative models show that the human cognitive system can form spatial memories that do not need to reflect the real world.

2.2.4 Visual, simulation, and analogical modelling

Visual modelling is very similar to narrative modelling except there may not be any explicit description of the surrounding context (scene). Such modelling is often said to use mental imagery, which refers to the mental images that people create in their ‘mind’s eye’ [Pyl73]. Mental imagery is used to perform a wide variety of everyday tasks such as answer a question regarding prior knowledge (for example, the shape of a dog’s tail) [GS11].

Experimental studies of mental imagery have investigated various tasks, including memorising a map, then timing imagined movements about the map, mentally folding paper to decide whether marked arrows will meet, deciding whether two shapes are the same after mental ro-
tation, and deciding whether a dot is superimposed over the previous representation of a letter within a grid [Pyl73, Kos87, Pyl02, GS11].

The ‘format’, or ‘representation’, of such mental imagery is a contentious issue that has further implications for what transformations are possible on information [Ner02, Tve05]. If such imagery has a picture form – similar to a photograph, or a hand drawn animation – while the individual may make inferences about the content of the picture, they should not be able to manipulate the contents of the scene. In contrast, if such imagery has a spatial form – similar to a computer generated 3D animation – the contents of the scene may also be mentally manipulated, and the individual should be able to reason about objects that are hidden from view. While the exact mental representation of such imagery has been disputed, it is likely that it is manipulated within either the visuo-spatial sketchpad component of working memory or the episodic buffer [KP11].

Simulation modelling, also referred to as visuo-spatial reasoning [Tve05], is very similar to visual modelling except that the individual also invokes semantic knowledge such as the force of gravity, or momentum, to consider how objects within a simulation may change or affect one another [Ner02, Tve05]. Analogical modelling involves trying to understand one process, by using the analogy of another process. For example, the flow of electrons through a conducting material is often explained using the analogy of the flow of a river [Ner02].

2.2.5 Human reasoning

The most abstract mental models are those used for reasoning [JL80, JL10]. The types of mental models investigated in human reasoning tend to emphasise the use of language, semantics, and logic to work out problems; and therefore may be described as ‘propositional’. However, it is stressed that mental models used during reasoning are not limited to propositional models, and that the type of mental model used depends on the task at hand, and also the individual.

2.3 External representations of mental models

As they become more abstract, it is helpful to represent mental models externally by producing a diagram, drawing, or written description. In fact, many professions, such as architecture, design, and programming, are very reliant on the production and manipulation of such external representations. The use of external representations is referred to as external cognition, but while it is has been extensively studied, the field has been criticised for being poorly integrated with similar research, and lacking an overall research framework [SR96].

Typically, such research has involved two types of study: in the first type, participants interpret presented representations; while in the second type, participants construct their own representations [Cox99]. Unfortunately, the results of such studies can easily be confounded by the skill and experience of the participants. In particular, it is difficult to determine whether the
external representation really does complement an internal mental representation, or whether the participants are able to translate the external representation through previous exposure to a similar representation – this is referred to as the resemblance fallacy [SR96].

In order to better understand how different representations can affect learning, studies have investigated providing participants with different types of external representations. While they have found that different representations provide differing levels of support for learning, they were not able to formalise a rigorous theory to explain why one representation was better than another [SK08]. Much of the research into external representations has been related to learning – recent research has tended to focus on the advantages of new technology such as multimedia and virtual reality.

2.4 Summary

Humans are provided with a memory system that includes a volatile short-term memory and a long-term memory that is effectively permanent. In the context of cognition, short-term memory is referred to as ‘working memory’ and contains a number of semi-autonomous components that are able to process perceptual information, and trigger the retrieval of information from long-term memory. The episodic buffer is now viewed as the key memory store that is under conscious control; and it is thought that it is able to chunk information from both short and long-term memory into perceptual objects. Long-term memory stores at least three different types of information: semantic information, episodic information, and spatial information.

A mental model represents the way that working memory and long-term memory are used during cognition, as well as how domain knowledge is stored in long-term memory. Mental models exist on a continuum between a reflection of the real world to abstract models based on language and rules. The type, or types, of mental models used during cognition depend on the task at hand. As they become more abstract, mental models become reliant on being exported to an external representation – the use of which is described as external cognition.
Chapter 3

Programmer cognition

Programming has long been recognised as a cognitively challenging activity, and, as such, has been extensively investigated within the overlapping fields of software psychology [Shn80] and the psychology of (computer) programming) [Wei71]. Early work investigated program comprehension and composition from the cognitive psychology perspectives of human problem solving [Bro75, Bro77], and human reasoning [SM75, Let86, LPLS87, Pen87], however, more recently, researchers have investigated programming from the perspectives of mental imagery [PB97, PB99], working memory [Dou08], and neuroscience [Par10]. While leveraging spatial memory has been described as the motivation for several software development tools [CM02, DCM+06, DR10], such papers rarely reference the cognitive psychology literature. Surprisingly, there are few (if any) contributions that explore how programming might be affected by the constraints of working memory as we currently understand them. This chapter reviews research that investigates programming from the perspective of cognitive psychology. Section 3.1 reviews Brooks’ influential model that treated programming as problem solving. Section 3.2 reviews the extensive literature investigating the cognitive models used during program comprehension. Section 3.3 reviews the modest amount of literature that has investigated the use of mental imagery during programming. Section 3.4 reviews research that has investigated the use of spatial memory. Section 3.5 discusses research related to source code as an external representation.

3.1 Programming as problem solving

An early cognitive model of programmers was inspired by the model of human problem solving presented by Newell and Simon [NS72]. Their model proposed that human problem solving involved three cognitive components: the information-processing system, the task environment, and the problem space. The information-processing system is the human cognitive capacity that interacts with short-term and long-term memory in order to solve a given problem. They proposed that the information-processing system might act as a ‘production system’ that contains chains of ‘productions’ – each of which containing a conditional and an action that is
performed if the conditional is true [Sim78]. The ‘task environment’ represents the constraints on how a problem is solved from the perspective of an independent observer. For example, the legal moves that may be made within a game of chess and the goal state that must be reached – check-mate or draw. In contrast, the ‘problem space’ represents all of the individual states possible based on the problem solver’s mental representation of the problem. The problem solver sequentially evaluates a series of states until they reach the solution state – though they may potentially be forced to return to a previous state, or start again. If a problem is presented in a different way, it may cause the problem solver to develop a different problem state, which may also affect the time required to solve the problem [Sim78].

Based on this model, Brooks describes programming as being composed of three processes: ‘understanding’, such as reading any provided requirements until an understanding of the problem to be solved is achieved; ‘planning’, which involves determining what programming components (or plans) are required to solve the problem; and ‘coding’, which involves translating the plans into source code [Bro77]. The coding process is described as using a production system that converts programming plans into source code statements within short-term memory that are then written out as source code. Brooks identified two other long-term memory structures that were required: the MEANINGS structure, which contains a set of attribute-value pairs that records the purpose of each variable and expression; and the CODE structure, which records how to access external memory in the form of code the programmer has already written.

Brooks’ work diverges quite significantly from the concepts presented by Newell and Simon in that there is no discussion – in the context of programming – of a ‘task environment’, nor is there any discussion of the ‘problem space’. This is to be expected as programming is a much less structured activity that, in contrast to games such as Chess, usually only has one participant, and only requires that text be added to a code editor such that at some point in the future it should conform to the syntactic rules of the compiler. In turn, this means that the problem space for programming would be effectively infinite.

3.2 Software mental models

Contemporaneous with Brooks’ model, Shneiderman published a similar model that sought to answer the question “what kind of knowledge (or cognitive structures) are available to the programmer in long-term memory, and what kind of processes (or cognitive processes) does the programmer use in building a problem solution in working memory?” [SM75, p. 5]. He answers that question by proposing that programmers have access to two types of knowledge: semantic knowledge, which “has to do with general concepts important for programming but which are independent of any specific programming language”, and syntactic knowledge, which “is more precise, detailed and arbitrary (hence more easily forgotten)”, and that it concerns details such as “the format of iteration, conditional or assignment statements, valid character sets or the names of library functions” [SM75, p. 5]. Shneiderman noted that programmers comprehend
programs by recognising “the function of groups of statements and then piece together these chunks . . .until the entire program is comprehended” [SM75, p. 13].

Both Brooks’ and Shneiderman’s original models, as well as those that have followed them, have tended to focus on what information is necessary while programming, and how such information is stored in long-term memory. Most later research studied program comprehension (as opposed to program composition), but continued the methodology of investigating how programming knowledge is built up and stored within long-term memory. In particular, later models focused on the process used by programmers to familiarise themselves with an unfamiliar software codebase.

In the context of programming, von Mayrhauser describes a mental model as:

“...an internal, working representation of the software under consideration. It contains static entities such as text structures, chunks, plans, hypotheses, beacons, and rules of discourse. Top-level plans refine into more detailed plans or chunks. Each chunk, in turn, represents a higher level abstraction of other chunks or text structures.” [vM95, p. 45]

Whereas, a cognitive model describes “the cognitive processes and information structures used to form the mental model” [SFM99, p. 17]. Well-cited cognitive models of program comprehension are those of: Shneiderman and Mayar [SM79]; Brooks [Bro83]; Letovsky [Let86]; Soloway, Adelson and Ehrlich [SAE88]; Pennington [Pen87]; and von Mayrhauser and Vans [vMV93, vMV94] (see [vM95] for an excellent review). Most of these models have been developed using observational studies similar to those of Brooks.

While Pennington recognises that analysis of a program can yield different types of abstractions, such as program goal, control-flow, data-flow, and condition-action (state-transition) [Pen87], there is no discussion as to whether these abstractions make use of different types of mental model. Rather it is suggested that different combinations of these abstractions may be predominant in a programmer’s mental model of a program.

In summary, while there has been extensive research into the cognitive models of programmers performing program comprehension tasks, this research has not investigated the actual mental models used by programmers.

### 3.3 Mental imagery

There has been remarkably little investigation into the use of mental imagery while programming. Petre and Blackwell carried out interviews that attempted to elicit the mental imagery of participants [PB97]. Ten experts were asked to design a solution for one of a number of programming problems, or one of their own choice. They were told they were free of coding restrictions, and were not asked to implement the solutions as code. During each session, participants were occasionally prompted regarding what they were seeing, or similar questions.
After the session was completed, the experts were questioned about their previous responses. The participants used mental imagery to visualise various aspects of their design including architecture, data-flow, and execution, and described a variety of different mental expressions including visual, verbal, and sound elements. In fact, the range of imagery described matches the continuum of mental models described in Chapter 2. Petre followed this study with a questionnaire that prompted 63 respondents to indicate how they used LabVIEW – 32 of the respondents made statements similar to those expressed during the interviews [PB99].

A difficulty with researching mental imagery is that it can be extremely hard for people to actually describe what mental representation they are using. According to Petre and Blackwell, “Experts are well-known for rationalising their practice on-the-fly” [PB97, p. 113], and may also inadvertently claim to use imagery that they do not in fact use.

Mental imagery is also often mentioned in visual programming research [GG84, GN95, Bla01, NPC01], however, while claims are made that visual programming notations correspond to programmers’ mental imagery, this does not appear to have ever been rigorously investigated.

### 3.4 Spatial memory and navigation

It has only been suggested recently that the mental model of software structure built up during program comprehension may take the form of a cognitive map within spatial memory. Cox et al. [CFO05] refer to software having a spatial structure they call codespace. They propose that during program comprehension a cognitive map is generated referred to as a code-to-concept map that allows the programmer to navigate software. They observe that programmers navigate software by either scrolling or jumping – scrolling corresponds to moving about within an individual source file, while jumping corresponds to when the existing programming context is replaced completely by another. In particular, Cox et al. make a distinction between the classic concept of ‘beacons’ [Bro83] and source code landmarks, they write:

“Landmarks identify a significant positional location in codespace. In a source file, subroutines that serve a similar purpose are often grouped together and preceded by a comment (computer ignored annotation) that identifies the group’s purpose. This comment is a landmark that identifies the start of a subroutine group. Landmarks are used during codespace transit as a navigational element.” [CFO05, p. 101]

Spatial memory has also been investigated in a variety of contexts related to computer user interfaces. Czerwinski et al. have developed a 3D interface for managing bookmarks they called the Data Mountain [CVDRH99]. They found that participants were able to retrieve bookmarks represented using this interface just as quickly as when the participants had originally arranged them four months previously; they also found that, even though the participants preferred to retrieve the pages using thumbnail images, the participants could (if required to) retrieve the bookmarks just as quickly using spatial position combined with a textual title that only
appeared once the mouse pointer was hovered over the bookmark [CVDRH99]. Later studies have sought to identify how this performance related to whether the representation was 3D or 2D. It has been suggested that it may be more important that an interface is explicitly designed to support spatial memory than whether it has a 3D representation [Coc04]. Claims of a relationship between spatial memory and program comprehension are supported by empirical studies that have shown a relationships between spatial ability and performance of programming tasks [TBH+06, JB07].

3.5 Code as an external representation

Despite source code being an obvious external representation of the mental model of a programmer, I have been unable to find research that has investigated it as such. Any mention of source code as an external representation has often been in relation to justifying program visualisation or visual programming [Sta98]. When attempted, searches of the literature for the terms ‘source code’ and ‘external representation’ have mostly returned references to papers that seek to generate a further external representation from source code. In existing program comprehension research, the ‘external representation’ represents the information that needs to be comprehended, not an external representation produced from an internal mental model [O'B03]. It must be concluded that either researching source code as an external representation has been overlooked or that it has been considered to not warrant further investigation.

3.6 Summary

Initial, early research into programmer cognition was based upon the models of human problem solving from cognitive psychology. Later research focused on developing cognitive models of programmers that describe the cognitive processes used to create a software mental model. In contrast, there has been relatively little research into the use of mental imagery and spatial memory during programming, however, it has recently been proposed that programmers explore the spatial codespace of software, and produce a cognitive map of software referred to as the ‘code-to-concept’ map. Surprisingly, it appears that there has been little research into source code being an external representation of a programmer’s internal mental model.
Chapter 4

Software visualisation

Visualisation has long been used as an aid to programming. For example, a very early paper describes a program for generating printed flowcharts from source code [Hai59]. It was natural that when graphics workstations became readily available that researchers would begin investigating how they might be used to support program composition and comprehension using computer visualisation. Such visualisation has since branched into two fields: visual programming, which aims to support the program composition task through the use of graphical notations; and program visualisation, which aims to support comprehension of software written in traditional, textual languages. While together these fields are now collectively referred to as software visualisation, and numerous books have been published [CIL87, Shu88, Cha90, Gli90a, Gli90b, Sta98, Die07] that contain representative samples from each of these fields; the actual research areas have remained distinct and separated. Although Myers did publish a paper containing taxonomies of both visual programming and program visualisation – [Mye86] updated by [Mye90] – the taxonomies were conceptually disjoint. This is further exemplified by various classifications and taxonomies that only apply to either individual area: visual programming – [BB94, BG04]; and program visualisation – [Bro88b, PSB92, RC92, RC93, PBS93, MMC02].

This chapter provides an overview of software visualisation. Section 4.1 provides an explanation of the terminology that is used to distinguish between different types of software visualisation. Section 4.2 provides an overview of visual programming. Section 4.3 provides a comprehensive review of program visualisation.

4.1 Terminology

Visual programming has been defined as both:

“...the use of meaningful graphic representations in the process of programming.”
[Shu88, p. 9]

and:
“...the use of visual expressions (such as graphics, drawings, or icons) in the process of programming.” [Cha90, p. vii]

Given these definitions, program visualisation could be seen as a sub-field of visual programming, however the traditional usage of visual programming has excluded traditional textual languages.

Knight and Munro define program visualisation as:

“...a discipline that makes use of various forms of image to provide insight and understanding and reduce complexity of the existing software system under consideration.” [KM99, p. 2]

In the past, some authors have not distinguished between the terms software visualisation and program visualisation [KM01], however, software visualisation is now often accepted as including visual programming and other software development visualisations, while program visualisation does not (referring only to the visualisation of existing software).

Price et al. provide the following, well-cited, definition.

Software visualisation is “the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software.” [PBS93, p. 213]

However, when considered strictly this does not include visualisations intended for software development – according to the authors this definition only ‘weakly’ includes visual programming to the extent that such a graphical notation can be used for program comprehension. The following definition of Young and Munro is more in line with current usage:

Software visualisation is “that which encompasses all aspects of visually displaying a software system.” [YM97, p. 1]

4.2 Visual programming

The research field of visual programming investigates the use of graphical notations for programming instead of traditional textual notations. There have been few attempts to categorise visual programming systems. Authors have typically categorised visual programming systems based upon their graphical representation – systems have traditionally been divided into the three categories: diagrammatic; iconic; and tables or forms [Shu88, SC92]. A taxonomy was presented by Myers [Mye86, Mye90], however this only differentiated between visual programming and programming-by-demonstration systems. More recently, Bottoni has proposed the use of a suite of meta-models as a basis for classifying visual languages, but did not provide a
A full review of visual programming is considered out of scope here as this report is concerned with visualisation of textual languages.

4.3 Program visualisation

The research field of program visualisation investigates the use of visualisation for the understanding of both dynamic and static properties of software. Recognisable sub-fields of program visualisation include: algorithm animation, execution visualisation, what I refer to as software meta visualisation, and visualisation for software understanding.

Dynamic visualisations provide a graphical representation of an executing program or algorithm. An early pioneering work is the short film “Sorting out Sorting”, which shows colour animations of numerous sorting algorithms [BS81]. With the increased availability of computers containing advanced graphical capabilities, many projects sought to replicate such visuals by generating them in real-time using computers. The simplest dynamic visualisations provide a snapshot visual representation of a program’s data-structures as it is executing. More advanced algorithm animations provide smooth animations representing an algorithm’s conceptual transitions, while program state visualisations provide views of a program’s aspects beyond pure data-structures [SP92].

Static visualisations show graphical representations that do not rely on the execution of the targeted program. Software meta visualisations visualise the meta-information associated with software, such as the number and sizes of source files that compose a program. Recent systems also visualise software evolution by mining code history from source code repositories. In contrast, software understanding visualisations are intended to help the viewer comprehend the structure of software, and the semantics of its source code.

4.3.1 Algorithm animation

Algorithm animations are typically used for pedagogical purposes. Simpler, fine-grained visualisations that show the state of data structures after each program step may not convey the conceptual information necessary for a student to fully understand an algorithm. Algorithm animation systems were developed that attempted to produce a meaningful visualisation to express the conceptual information of an algorithm not possible by just showing individual program states. An example is a visualisation of sorting that shows bars being sorted by length.
It may not be obvious that two bars are being swapped if they are shown being copied to a temporary variable, however, if a smooth animation shows the bars swapping positions, the conceptual information is shown even though it might not reflect valid states of the program.

Brown university ALgorithm Simulator and Animator (BALSA) and BALSA-II systems are early examples of algorithm animation systems [BS84, Bro88a]. They are capable of showing animations of many different types of program tasks including sorting, tree-building, and compilation. They also support limited scripting, which allows animations to be pre-configured for later playback. Both systems work by incorporating system calls at algorithmically important points in the execution of code referred to as \textit{interesting events}. BALSA II [Bro88a], developed for the Macintosh, sought to make developing algorithm animations easier by providing a general framework and scripting functionality. BALSA and BALSA-II were succeeded by the Zeus system [BCA91].

Typically, regardless of the algorithm animation system used, each separate visualisation requires a lot of effort to produce. Attempts to reduce the effort required included providing direct manipulation interfaces for the specification of the visualisation – ALgorithm Animation Design and Description using INteraction (ALADDIN) [HHR89], Demonstration ANimation CrEation (DANCE) [Sta91], and A New Interactive Modeler for Animations in Lectures (ANIMAL) [RSF00]. Instead of annotating algorithm code, the self-animation approach (as it is referred to in [LST98]) instead encapsulates algorithm animation code within reusable libraries that provide \textit{visual data types}. Examples include: Animus, an algorithm animation system developed in Smalltalk [Dui87]; and Eliot [LST98].

In contrast, language-based approaches relied on the composition, or generation, of a script that would then be interpreted by the algorithm animation system. Transition-based ANimation GeneratiOn (TANGO) [Sta90] is a system and framework intended to ease creation of new animations by implementing the \textit{path-transition paradigm}, which describes an animation using four abstract data types – images, locations, paths, and transitions – it was succeeded by XTANGO [Sta92]. The Samba system [Sta97] extended this approach to have students construct algorithm animations themselves instead of watching already prepared animations. An advantage of the language/scripting approach is that regular software can be modified to generate such a script as it executes. Other systems that are intended to allow users create their own animations are MyJava [CS01], and Dynamic Algorithm Visualization Environment (DAVE) [VJ08]. Recently, systems have been developed that define animations using an XML driven approach, such as the Java-Hosted Algorithm Visualization Environment II (JHAVE II) [NMG07]. Cyber-Films [RM05] are an alternative approach where each film consists of scenes and frames that may be viewed in a non-linear order. The Win Hope Integrated Environment (HIPE) system [NBPFVI00] integrates algorithm animation into the HIPE functional programming environment.

Generally, algorithm animation systems have created hi-fidelity animations, an alternative approach has been to investigate whether low-fidelity algorithm animations may be just as
effective – the Spatial Algorithmic Language for StoryboArding (SALSA) system provides a simpler language that allows animations to be created more quickly than otherwise [HD02]. In contrast, motivated by a survey that found that most algorithm animation systems sought to visualise novice level algorithms, the recent Vamonos [CR15] system provides visualisation for more complex graph algorithms.

Due to the desire to run algorithm animations on multiple operating systems, many recent systems have been developed in Java. In particular, several systems have been developed to run over the web as Java applets – for example, Java And Web-based Algorithm Animation (JAWAA) [PR98, AFJ+03]. The Web Generalized Algorithm Illustration through Graphical Software (WebGAIGS) system [NB98] runs within a web browser but also supports multiple views that can show different aspects of the running algorithm. The Web Algorithm Visualization Environment (WAVE) [DFL00] uses a publication-driven approach whereby a public blackboard is updated each time data is modified by a program. Other web-based systems include: Leonardo Web [BDF+05], and a pure HTML and Javascript web-based system [Kar09] that uses source animations that are specified in the eXtensible Algorithm Animation Language (XAAL) format [Kar05]. The original JHAVE system [NEN00] is a client server architecture into which more specific algorithm visualisation engines may be plugged.

### 4.3.2 Execution visualisation

*Execution visualisations* are used to visualise aspects of a running program. The simplest examples of these are mainstream debuggers that show the values of variables currently in scope, and allow data-structures to be traversed.

Program state visualisations [SP92] are used to visualise more abstract aspects of a running program such as representing the control-flow using a call graph, or showing a view of the runtime stack. These visualisations typically show some mix of the code, data-structures, and control-state of the program. These visualisations can be used for debugging (checking that the execution is occurring as intended), or for comprehension by watching the running program to better understand how it works. Two reviews of the area are [Rei07] and [RT11].

When executing software, it is often important to understand how much of the program is actually executing (coverage), as well as what parts of the program are executing predominately (and therefore are candidates for optimisation). Several systems have been developed that provide the user with insight into how a program is executing – the Graphically Interacting Program Monitor [CR83] represents program execution using animated Nassi Schniederma diagrams; LOGOmotion [BB90] provides a number of views including a stack view of execution and trace views; VOGUE [Koi93] uses a 3D representation to visualise properties of parallel systems – for example, showing the execution pattern of a number of different processes; JInsight [DPV98, DPJM+02] provides a reference pattern view, an invocation browser, and an execution pattern view; BLOOM [RR03] is a general software visualisation framework that also
supports execution visualisation through the visualisation of execution traces; JIVE [Rei03] is a related system for visualising trace information; Tarantula [JHS01] is a system that depicts the execution of a program as it is tested; Javavis [OS02] is able to show object diagrams and sequence diagrams of running programs; EVolve [WWB+03] is a general purpose framework for visualising runtime information; OGRE [MR04] represents objects as interconnected boxes within a dynamic 3D scene; VILLE [RLKS07] is a language independent system that can visualise the execution of a program written in two different languages at once. DYVISE [Rei09] provides a visualisation of the Java heap; ExploreVis [FWWH13] uses the 3D city metaphor to visualise traces for large software landscapes; and The Brain [PJ13] is an execution visualisation inspired by visualisations of neurons firing.

Due to the desire to visualise the potentially complex data structures that can be dynamically built during the execution of a program, several systems have been developed that present various dynamic aspects of running systems: Incense [Mye83] is able to graphically represent data structures such as variables, arrays, objects, and pointers; PV [BCH+85], is able to represent an architecture diagram as well as some data structure diagrams; PECAN [Rei85] is able to show expression trees, data type diagrams, flow graphs, and the symbol table; Amethyst [MCS88] provides default displays for data structures to reduce customisation effort; Aktri [RD88] draws data structures as they might be drawn in a textbook; The University of Washington illustrating compiler [HWF90] is able to represent several data structures including graphs; Jeliot [MMSBA04] is a development environment for novices that provides a fully automatic visualization of data and control flows; jGRASP [CJH+07] is able to display dynamic object viewers for data structures such as binary trees; and PCVis [KS13] is a tool that shows the in-memory composition hierarchy of objects.

Sometimes, the most effective way of finding a bug in software is to step through the code line-by-line using a debugger – Zstep shows the code being stepped through in an editor that substitutes values into expressions [Lie84].

Concurrent, multi-threaded, and parallel systems are difficult to develop because of non-deterministic access to memory by different actors in the systems. Several execution visualisation systems have been developed to try and provide programmers with more insight as to the interactions between these actors – Dyview [RK10] is a system for visualising threads, transactions, and tasks; SyncTrace [KTD13] is a visualisation system for analysis of threaded programs that is based on activity diagrams; and Synchrovis [WWF+13] uses the city metaphor to visualise the execution of threaded programs.

4.3.3 Meta visualisation

Software meta visualisations graphically represent information about programs. Early systems represent information such as average file size, the number of functions in files, and computational complexity. Later systems provide visualisations of software structure either in the
form of 2D graphs or 3D scenes. The main differentiator between meta visualisations and the software understanding visualisations described in Section 4.3.4 (Software Understanding) is that meta visualisations are not intended to support the comprehension of source code.

SeeSoft, an early example of meta visualisation, represents each of a program’s source files as a vertical column-bar within which each line of code is represented by a coloured horizontal line. The colour of these lines can be changed to convey information regarding that line, e.g., that it has been modified due to a specific modification request [ESSJ92]. This work was soon extended by a space-filling visualisation, SeeSys, that is able to show the distribution of files amongst directories that belong to different subsystems [BE95]. Each subsystem is represented by a square that has a size proportional to a chosen software metric – such as lines-of-code, or code complexity. A subsystem can be zoomed into to just show metrics associated with those directories and files, and specific files can be zoomed into to show the actual source code [BE96]. The SeeSoft metaphor was later applied to produce a three-dimensional meta visualisation. FileVis [YM98], Imsovision [MLMD01], and the Source Viewer 3D (sv3D) framework [MMF03, MFM03b, MFM03a] represent files as 3D shapes within 3D environments. The sv3D system represents files as 3D containers, which hold poly-cylinders that each represent a line of code. This framework was later integrated with the IRiSS system to provide support for concept location [XPM06a].

Other tools have investigated using various 2D or 3D graphing methodologies to represent software structure or metrics, such as: the Generic Software Exploration Environment (GSEE) [Fav02], which displays source code structure in a mix of tree and graph representations; Software Landscapes [BD04], which represents software in nested spheres; CodeCrawler [Lan04], which shows various software metrics in a number of different forms; and Voronoi Treemaps, which shows software structure using the tree-maps graphing technique [BDL05].

Several systems use the city metaphor [PBG03], which shows source code in the form of an aerial view of a city – the Imsovision system [KM00] was developed with virtual reality in mind so that people could walk through the landscape. In such visualisations, generally each building represents an individual class, and neighbourhoods may represent packages. Attributes of buildings, such as colour, building footprint, and height, may be mapped to source code attributes such as complexity, the number of class attributes, or the number of methods within a class. The CodeCity system depicts object-oriented software as “habitable cities that one can intuitively explore” [WL07, p. 92]. EvoSpaces is a similar system whose building types distinguish the types of files represented [AD07]. The CodeMetropolis system uses the popular MineCraft game engine to visualise software and facilitate collaboration [BB13]. The SkyscrapAR system represents the city metaphor using an augmented reality system [SSMM12]. A recent project has gone one step further and has 3D printed objects to represent 3D code structures in physical form [FKH15]. As a complaint regarding city metaphor visualisations is that the placement of source code artefacts tend to have little meaning, the Software Cartography system [KELN10] positions software artefacts to reflect the vocabulary that is used within the
Other systems have focused on visualising revision control information. Some show information such as the change in the number of source files of a system over time [GT01]; others use a variation of the SeeSoft metaphor to show changes in code over time [FD04, VTvW05]. The GEVOL system [CKN+03] represents a system’s inheritance hierarchy as a three-dimensional coloured tree that shows changes to the hierarchy, and by whom. Other three-dimensional systems have represented files as cubes arranged in a series of two-dimensional, x-y, planes along the z-axis, which represents time – Visual Revision Control System (VRCS) [KC97] and Software Release History [GJR99]. Similar systems include: softChange [GH06], and BugCrawler [DL06]. The CVS Viewer 3D (cv3D) system represents software change history using visualisations similar to sv3D [XPM06b]. The Churraesco framework [DL08] provides a web interface for software evolution.

Several software evolution tools use the city metaphor to represent code changes. The CodeCity system was extended to also show software history [WL08]. The spatial stability of the resulting visualisation as software changes is an important issue for such systems – i.e., after source code changes, ensuring that an unmodified class remains represented in the same spatial position if the visualisation is regenerated. The Evo-Streets system aims at “providing stable software cities without knowing the development history in advance” [SL10].

4.3.4 Software understanding

Software understanding tools are intended to aid the comprehension of existing software. These tools can be divided into two conceptual categories: enhanced representation; and software exploration tools.

Enhanced representation tools provide an enhanced visual representation of source code, typically aimed at the code level. Pretty printing tools enhance source code by using typography to emphasis the structure of source code – such as presenting keywords in bold text. An early example of this type of tool is the SEE Program Visualiser [BM90]. Other tools generate graphical representations, such as flow charts, from source code. The earliest example of this is FlowCharter [Hai59]. Other tools that generate control-flow diagrams are: First Programming Language (FPL), Pascal/HSD, Programming-Support System, PIGS, and Pigsty (all described in [Shu88]), as well as GRASE [AR84]. More recent tools call themselves companion visualisations as they provide an auxiliary visualisation that is intended to be viewed alongside the existing text. The Control Structure Diagram (CSD) [CIHM98] provides a graphical annotation in the margin of an otherwise traditional text viewer. Other tools, such as PegaSys [MH85], show data-flow.

Software exploration tools provide the ability to view a program at the macro (module or class) level, then drill down to study the micro (code) level. These tools are intended to aid program comprehension and understanding during maintenance. Program-slicing [BG96] tools
allow the viewer to extract from a program the function calls, or program statements, that are associated with a specific control-flow, or computation. Other tools allow the viewer to better understand the call graph of a program – Ghinsu [LA93]; CodeSurfer [AT01].

Numerous tools provide a diagrammatic overview of classes and methods that aims to support the programmer’s software mental model. Rigi [MK88] allows different aggregation hierarchies to be displayed within separated windows. The Software Landscape [MHP93] allows the user to navigate into projects and around entities created during software development. Javazoom [Hei98] provides a class diagram that represents methods as purple boxes within green squares that represent classes next to a code viewer. Whorf [BGSS92] is a hypertext based environment that presents different types of views within windows, such as a variable cross reference, a function cross reference, a call structure view, and a source code view. VIPR [CSZ96] recursively represents classes as circles that may contain further class circles – as inner classes may become quite small VIRP allows the user to zoom into inner circles. Portable Bookshelf (PBS) [Sim98] is a web-based environment for representing and searching software artefacts. Simple Hierarchical Multi-Perspective (SHriMP) [SBM+02] represents Java packages, classes, and methods, as boxes that may be zoomed into to reveal more detail – additionally dependencies between these are indicated with arrows. Creole [LMSW03] is an Eclipse plug-in that provides some of the functionality of SHriMP. Relo [SKM06] allows the user to explore software using a UML-like representation. Dependency Analysis for Java (DA4Java) [PGKG08] is a dependency analyser that is similar to Creole but is able to reduce the number of irrelevant classes and connections from its representation. CodeSonar [RS07] provides an auxiliary view for Eclipse that helps visualise source code and quality defects. The View Infinity tool [SFA+11] provides seamless and semantic zooming of different abstraction layers of a software product line (SPL). Java Program Dependence Graph (JavaPDG) [SSHP13] shows a dependency graph next to a source code viewer. SourceMiner is a software visualisation engine and associated views for the Eclipse IDE [dFCdMN13]. Softwarenaut [LLN14] is a visualisation tool for architecture recovery. VISCTE [SA16] shows an automatic breakdown of components in a system. GrepCode [Gre] is an online website for navigating through the source code of several pieces of open source software – direct navigation is supported via various auxiliary views including a file browser; relative navigation is supported through hyperlink-enabled method calls and type references.

4.4 Summary

Software visualisation collectively refers to the fields of visual programming and program visualisation. While visual programming focuses on the creation of novel programming notations and environments for developing software, program visualisation focuses on providing tools to aid the comprehension of existing software, or algorithms, written in traditional programming languages. While neither of these fields is focused on producing novel software development en-
environments that support existing textual programming languages, they both provide guidance and inspiration for features novel development environments might provide in the future.

This project is conceptually similar to software exploration tools in that it aims to develop a tool that supports the formation of a programmer’s mental model. However, even though they attempt to increase visual momentum, most software exploration tools share the same conceptual problem as mainstream development environments – after each navigation the existing programming context is obscured by the next – negatively interfering with visuo-spatial memory.
Chapter 5

Software development environments

Programming tools have progressed significantly since the advent of computers. Initially, programmers had to use a collection of stand-alone tools to develop software that were collectively referred to as *programming environments* – a contemporary example would be the use of the GNU Compiler Collection (GCC), the GNU DeBugger (GDB), and a stand-alone code editor. In order to make software more portable between different machine architectures, interpreted programming languages were developed that only required the interpreter be ported to different systems. Later, some of these – such as Interlisp [TM81] and Smalltalk [Gol84] – also incorporated code editors of varying sophistication. The Smalltalk 80 environment, developed at Xerox, is acknowledged to be one of the first development environments to feature a graphical user interface (GUI).

Due to vendors such as Microsoft and IBM not shipping development tools with their operating systems, third-party vendors such as Borland developed and sold *integrated development environments (IDEs)* that integrated programming environment functionality into stand-alone applications. Recently, in order to better control the platforms they provide, operating systems vendors have supplied their own development environments that are better able to leverage new operating system features. The free Xcode IDE quickly became the de-facto development environment for the Apple Macintosh when Apple transitioned from the PowerPC processor to those supplied by Intel, and Microsoft’s Visual Studio has become the default IDE for development on the Windows platforms. The exceptions are the IDEs for the Java programming language, and environments that have focused on supporting niche programming languages or technologies. While most of these environments include advanced capabilities – especially in the areas of re-factoring software and debugging support – the paradigm has remained predominantly that of a whole source file editor. In contrast, within the research community, numerous experimental software development tools and environments have been developed that feature novel programming interfaces.

The following sections describe both research development environments, as well as a representative sample of notable mainstream development environments. Section 5.1 describes early research environments. Section 5.2 describes contemporary research environments. Section 5.3
describes notable mainstream environments.

5.1 Early research environments

During the 70’s, due to the success of high-level programming languages, researchers began investigating ways to generate programs directly from high-level specifications – a field referred to as automatic programming. A well-known example of such a system is The Programmer’s Apprentice [RSW79, RW87b, RW87a, RW88]. However, as it was later realised that such systems only suited domain specific fields where expert knowledge had been previously captured [RW88], and as such systems did not intend to modify the programming interface, they won’t be reviewed here.

As early computer science research focused on the syntax and semantics of programming languages, other environments sought to aid program composition at this level. Syntax-directed editors were developed to help programmers avoid syntactic mistakes during programming. These systems later grew into multi-view systems that were developed to help programmers better understand their programming context. It is important to remember that often these editors were limited by the computer hardware of the day.

5.1.1 Syntax-directed editors

As a significant problem with programming is the accidental introduction of syntax errors, syntax-directed editors were developed that forced the programmer to always have a program with correct syntax. The DIALOG system [CEL67] provides an interactive display that allows the programmer to compose a program line by line and ensures syntactic correctness by restricting the characters the programmer is able to select. The Emily system [Han71] forces the programmer to create a program as a tree structure that complies with the syntax of the programming language – the system displays non-terminal nodes that must be replaced with syntactically correct identifiers in order to complete the program. Similar systems that rely on tree composition are MENTOR [DGHKL80], the Incremental Programming Environment (IPE) [MMF81], and a GRaphical Syntax-directed Editor (GRASE) [AR84]. In contrast, the CAPS system [WDT76] provides the user with an editor that ensures that all text behind the cursor is the prefix of a legal program.

While syntax-directed editors also formed the basis of more extensive environments such as the Cornell Program Synthesizer [TRH81], PECAN [Rei84a], and Gandalf [HN86], experience revealed that programming by constructing programs that were always syntactically correct was not natural for programmers and that it would “often get in the user’s way” [Rei99, p. 336]. Programmers preferred to use text editors that would simply warn them of any syntactic errors like the Interactive Programming Support System (IPSS) [BMP68].

As syntax-directed editors rely on having access to a codification of a programming lan-
guage’s syntax, editor generator systems were developed to tailor these systems for different languages. The Synthesizer Generator [RT84] was developed to generate variants of the Cornell Program Synthesizer. Similar systems are PECAN [Rei84a, Rei84b], and Gandalf [HN86].

The development of code editors that enforce a syntactically correct structure has been continued by those researching development environments for novice programmers – recent environments are reviewed in Section 5.2.6.

5.1.2 Integrated environments

Syntax-directed editors grew from being substitutes for editors to being full integrated development environments. The Cornell Program Synthesizer [TRH81] is able to execute syntactically complete programs, trace the flow of execution, monitor variables, and either slow down, or step through, execution.

As bitmapped, graphical displays became available, software development environments became more sophisticated. Interlisp [TM81], Smalltalk [Gol83], and PECAN [Rei84a] make use of graphical displays by providing multiple views of software. The PECAN system [Rei84a, Rei84b] provides a syntax-directed editor, a declaration editor, and a flow graph editor. Such systems also incorporate views that display runtime information such as the contents of the stack during execution. PECAN was succeeded by the GARDEN conceptual programming environment [Rei87], the Friendly Integrated Environment for Learning and Development (FIELD) [Rei90, Rei95] – later commercialised as DEC-FUSE [HL95], and the DESERT environment [Rei96, Rei98, Rei99].

5.1.3 Call-based navigation

Several early programming environments present a view of the call graph. GELO [RMD89], which was integrated into the FIELD environment [Rei90, Rei95], is able to show a dynamic view of the call graph of a program. Similar capabilities were included in the DEC FUSE environment [HL95], which was an evolution of FIELD. Recently, such views were also made available in BLOOM [Rei01, RR03].

5.2 Contemporary research environments

Recently, there has been renewed interest in the development of novel programming interfaces for software development environments. These systems may be categorised as: spatial navigation environments, which attempt to provide a visible representation of an entire codebase in order to leverage spatial memory; relative navigation environments, which provide additional support for navigating the call graph of a program; recommender systems, which use prior navigation history or heuristics to recommend files that may be relevant in order to ease navigation; working set environments, which provide increased support for working with a set of
files, classes, or methods; and code augmentation environments, which augment source code with diagrams allowing code complexity to be better understood.

5.2.1 Spatial navigation environments

In mainstream development environments, direct navigation is often achieved by using an auxiliary representation of the source directory hierarchy. While some environments list the methods available within a class, others require the programmer to scroll through the file to find a desired method. Apart from the time taken, such navigation also assumes that the programmer can remember the names of the package, class, and desired method. In contrast, spatial navigation environments provide a map-like layout of source code on a 2D plane so that the programmer can find a method “perceptually rather than cognitively” [DeL05, p. 309]. These systems seek to reduce disorientation by making navigation quicker and less of a cognitive activity. The Software Terrain Maps system [DeL05] presents a map of a codebase that is inspired by a cartographic map. In one manifestation, each method corresponds to a region of a hexagonal grid, where the size of the method’s region is proportional to the method’s textual size. As method positions only change as the codebase changes, the programmer may select a desired method by selecting the map area corresponding to the method. To keep the programmer oriented, the authors suggested that recently accessed methods could be highlighted with a vapour trail [DeL05]. Code Thumbnails [DCM+06] and Code Canvas [DR10] present a more traditional representation in the form of a thumbnail view of a codebase. The Code Thumbnails Desktop displays each source file as a thumbnail image that can be selected to take the programmer to the indicated file. As the arrangement of thumbnails is spatially consistent, the programmer may utilise spatial memory to find the location desired. Code Canvas [DR10] extends the Code Thumbnails concept by placing the code thumbnails on a desktop that supports semantic zoom, which shows “different levels of detail at different levels of zoom” [DR10, p. 208].

5.2.2 Relative navigation environments

Environments that support relative navigation allow the programmer to traverse between source code file locations using programmatic links such as method calls. Two plug-ins for the Xcode environment have been developed that support relative navigation: the Stacksplorer system [K+11] allows the programmer to easily navigate to methods above or below the current method in the call graph by providing clickable representations of those methods in lists on the left and right sides of the window, respectively; and the Blaze system [KKKB12] shows an entire call graph branch in a view to the right of the source code editor that uses a combination lock metaphor, which allows the user to select different possible branches by clicking left and right arrows presented for each method of the call graph. In contrast, the Fluid Source Code Views system [DSE06] allows the contents of called methods to be expanded inline within the calling method. This reduces thrashing between files as such behaviour is often due to an attempt to
simultaneously compare a calling method with a called method. Prodet [AFQ+15] is a software navigation tool that provides a search view, a navigation view featuring a call graph, and a map view. Selecting a suggestion from the search view causes the navigation view to show the call graph surrounding the selected method, which can then be explored – selecting a method centres it in the view. The map view shows the classes and methods that have been navigated through.

5.2.3 Recommender systems

Recommender systems provide navigation suggestions based on either prior behaviour or analysis of the codebase. These systems seek to reduce disorientation by reducing the number of classes navigated. The NavTracks system [SES05] creates a model of relationships between files based on prior navigation history then unobtrusively suggests files of immediate interest using a Related Files view. Mylar [KM05] is a related system that uses a degree-of-interest model to drive filtered views that highlight likely task related elements. The REACHER [LM11] system presents a graph based on a semi-automated traversal of the local call graph.

AutumnLeaves [RND09] takes the opposite approach, instead of recommending files that may be of use in the future, it recommends files that it determines are unlikely to be used in the future by closing or greying out the window or tab.

5.2.4 Working sets

Software navigation is often for the purpose of finding code fragments that are related to the current task – this is referred to as a working set [B+10], or alternatively as a task context [K+06]. Environments that support working sets allow the programmer to group arbitrary sections of source files together that are related to a particular task. These systems seek to reduce disorientation by eliminating the need for navigation.

Knuth’s literate programming system, WEB [Knu84], might be viewed as a pre-cursor to working set environments as it emphasises creating literate source where ‘concepts’ are introduced in an order that is best for human understanding. The Sheets Hypercode Editor [SK97] provides similar capabilities through the use of file-like ‘sheets’, which contain linear groupings of arbitrary code fragments. The Code Bubbles environment [B+10, RBLV14] provides a virtual workspace upon which methods may be arbitrarily placed (as bubbles). Called methods may be ‘budded off’ adjacent to the calling bubble, allowing the programmer to view an arbitrary number of methods simultaneously. As the workspace is many times wider than the display, multiple bubble groups can be arranged for different tasks. The Patchworks system [HF14] presents a 3x2 grid of ‘patch’ editors that can be used to edit files, classes, or individual methods. I categorise the Patchworks system as a working set system rather than a spatial navigation system because the grid of patches must be manually populated, and only 6 patches can be viewed at once.
5.2.5 Code augmentation

During programming, a programmer’s preconception of what a piece of code does is often different to what the code actually does. Code augmentation systems present a diagram that provides a visual cue regarding control flow characteristics. GRASP [HC95] – now reimplemented in Java as jGRASP [HCIB04] – augments source code with a Control Structure Diagram (CSD), which graphically highlights control flow aspects such as conditionals and loops. GRASP is also able to generate a UML view that can be used to navigate source code files, as well as a code complexity view that shows the complexity of code.

5.2.6 Environments for novices

As environments for novices are considered out of scope for this thesis because they do not need to address navigation issues, this section only includes an overview of recent environments – for a detailed review of earlier work refer to [KP05].

Novice programming environments may be targeted at specific domains – such as the Alice environment [CAB+00], which targets programming 3D animations; or as general-purpose programming environments intended for novice users. Such environments often share the characteristic that users program by selecting programmatic templates from a palette of elements, which are dragged to a program construction area, however, in contrast the HANDS (Human-centered Advances for the Novice Development of Software) environment [MPK04] has been developed to investigate the use of programming using natural language.

Alice is an influential programming environment for creating 3D scenes that is aimed at people with no 3D graphics or programming experience [CAB+00]. Alice allows the user to place objects within a scene, such as a person or a rabbit, and then program the scene by dragging commands from a palette to a programming area. Items in the programming area are applied to the object that is currently selected. The system also allows the user to specify whether actions occur in parallel or in series.

The Greenfoot environment [KÌ0], is a Java development environment aimed at teaching programming to young novices. Like Alice [CAB+00], and the later Scratch environment [MPK+08] (discussed below), Greenfoot provides the user with a world and actors that can be manipulated programatically. Users are provided with an already completed example that allows them to add existing actors to the world and configure them using contextual menus. Only once a user is comfortable with the higher level mechanisms, are they introduced to the programming aspects. Unlike Alice and Scratch, Greenfoot only provides a standard code editor for Java.

Other environments that incorporate the metaphor of controlling scenes include: Toque [TSD+10], which uses a cooking metaphor and has users control a chef to create a dish; Kodu [Mac11], which is a language and environment designed for young children that is available on the Xbox 360; and CodeSpells [EFG13], which uses the metaphor of a spell book that allows
the user to both learn new classes representing spells, and modify existing spells.

A large number of recent environments for novices have been inspired by the LogoBlocks environment [Beg96], which was itself inspired by the earlier Logo Turtle system [Pap80]. Referred to as *blocks-based* environments, these systems provide a palette of *blocks* representing programmatic statements, which can be dragged onto a main work area. Each block has cut-outs similar to a jigsaw piece, which inform the user where pieces may be placed. Scratch [MPK+08] is an online (Flash-based) programming environment for producing animations. Much of Scratch is purpose built for developing animations, however, a user can also use a blocks-based visual code editor to specify how objects are manipulated. AppInventor [Wol11] is a block-based environment for programming Android mobile devices that provides access to high-level services such as SMS, and GPS location. Blockly [F+13] is a visual programming editor implemented in Javascript, which has been released for use in web projects by Google. As the visual code is built up within the visual code editor it is also shown in textual form on the right-hand side of the screen in a programming language selected by the user. Tiled Grace [HN13] is a web-based development environment for the Grace programming language that can represent code either in a visual blocks-based metaphor or as text source code.

The frame-based editor for the Stride programming language [BAK84], which is distributed within the Greenfoot environment [K10], is an attempt to get the best out of both block-based and text-based development environments. Each editor represents an individual class source file similar to mainstream environments, which makes it suitable for non-trivial projects. Like other block-based environments, programmatic blocks, such as if or while statements are represented as individual entities within an editor as *frames*, however, rather than needing to be dragged from a palette, which can be tedious and time consuming, new frames are added at a *frame cursor* by pressing keyboard characters when a frame cursor is active. Frames include *frame slots*, which hold sub-frames, and *text slots*, which take normal program text. The end result is that the frames-based editor always ensures the syntactic validity of block statements all the way up to the class level.

### 5.3 Notable mainstream environments

Many code editors and development environments have been developed that each provide a different set of features. As describing them all would be beyond the scope of this section, a representative sample of popular environments is reviewed focusing on how source code is represented and navigated. First, a description of notable features is provided, then each environment is discussed. Table 5.1 provides a breakdown of which features are support by which environment.
5.3.1 Code representation and navigation features

Code editors provide features that are directly related to viewing and editing source code, while development environments also provide features reliant on the ability to understand the source code at semantic level. While the features provided by environments are often similar, there do tend to be subtle differences between how each environment implements a feature.

Environments provide different levels of support for allowing a user to continue their work at a later date. Those that provide *project* functionality save a collection of directories or source code files that contain or represent the source code of a project. This allows a user to later reopen the project and then open relevant files. Environments that provide *workspace* functionality save the state of the workspace so that any files are reopened to the state they were in when the workspace was closed.

Many environments allow multiple editors to be opened by using the tab metaphor. Environments that provide *tabs* let the user click (or use shortcut keys) to navigate between a set of code editors – selecting a different tab causes the existing editor to be obscured. Environments that provide *tab groups* allow the user to have multiple code editors displayed simultaneously, and usually allow tabs to be dragged between the individual tab groups. Environments that implement a *splittable code area* allow several code editors to be displayed simultaneously by splitting the screen either horizontally or vertically. Some environments also support *splittable editors* so that two different locations in a source file may be viewed at once – others provide a similar capability by displaying a code buffer in multiple editors. Environments that support *multiple-windows* usually do so by allowing a tab to be dragged out of the existing window to create a new window.

Recent environments have begun providing additional information regarding the content, or the status of the content, of a source code file. Many such environments now provide coloured marks on the scroll bar, which indicate points of interest such as syntactic errors – for convenience, these are referred to as *info scrollbars*. One of the environments reviewed (Sublime Text) provides a *mini-map* for each editor, which shows a thumbnail view of the code allowing fast navigation by selecting a specific file region.

Environments provide a range of features that allow a user to navigate between and within source code files. Environments that provide project support usually provide a file-browser auxiliary view that represents the file hierarchy. More sophisticated environments allow the user to navigate between files using relative navigations. Environments that support relative navigation allow the user to easily follow a method call to its definition – in these environments, holding down a modifier key and hovering the mouse over a reference causes it to turn into a selectable hyperlink that reveals the desired definition. However, some environments only support *indirect relative navigation* by providing ‘go to definition’ type functionality. Several environments provide an auxiliary *call hierarchy* view that can be refreshed to show the chain of methods that call the current method, or show methods called from the current method.
A common method of navigating is by searching the current file or all files for textual patterns. Environments that support find allow the user to navigate to areas containing text that matches the search pattern – typically by jumping (scrolling) through the file. In contrast, environments that support find in files usually open an auxiliary view that shows the lines that contain the search pattern for appropriate files. To provide increased support for developing a working set of files – some environments support marking file lines with bookmarks, which are then also displayed in an auxiliary bookmarks view, or may be accessed using a keyboard shortcut.

Development environments may also provide a number of auxiliary views whose contents are generated from a semantic understanding of the codebase. Environments that provide a class inheritance view are able to interpret the semantic connections between files and show the class inheritance hierarchy in a tree view. Similarly, some environments provide a class composition tree view. While earlier environments could generate a call graph, mainstream environments generally only provide the call hierarchy view that is local to the currently selected method. Finally, some environments allow the source code to be represented using the class browser paradigm, which represents source code using individual method editors – a method is selected by choosing the package, class, and member/method from contextually sensitive auxiliary lists. While some environments do provide dropdown lists that are conceptually similar, these are not considered to meet the requirements for a class browser view. The Code-flow software exploration tool was designed to present a class browser style interface – refer to Chapter 10.

5.3.2 Code editors

Code editors, as distinct from development environments, support the composition and modification of source code but do not incorporate, or provide access to through external tools, either interpreters or compilers that allow such code to be executed. In practice, this can be a hard distinction to make as popular code editors often also provide scripting or plug-in capabilities that allow additional functionality to be accessed. Therefore this distinction should be taken only as intended – as a way to conceptually group these environments.

Sublime Text [Ski08] is a popular tab-based code editor that supports project and workspace management, as well as multiple windows. The code area is splittable and also supports tab groups. Each editor contains a mini-map that can be used for navigating files, and it also supports find and find in files. The environment does not provide any additional auxiliary views, but does support rudimentary indirect relative navigation.

The open source Vim [Moo91] and Emacs [SS76] code editors are popular because of their portability across operating systems and their flexibility. Both systems have variants that are run within a terminal or as desktop applications with additional features – for example, MacVim [Mac] also supports tabs. Both systems use the buffer approach where editor content is stored in off-screen buffers that may be displayed in any available code editor. Code editors themselves
are able to be split either horizontally or vertically to create a new adjacent editor. This flexibility makes it possible to arrange related source code in useful ways. A directory listing may be opened in an ‘editor’ allowing quick access to any contained files, or navigation to other directories. Navigation is mainly achieved within a file using elaborate keyboard shortcuts, and between files using search features, or set marks (bookmarks). Neither system provides additional auxiliary views for displaying software structure. As both systems are extremely extensible a discussion of further capabilities is out of scope.

5.3.3 Interpreter environments

VisualWorks Smalltalk [Cin99] is descended from the original Smalltalk 80 [Gol84] interpreter and development environment. These systems use a class browser interface that presents Smalltalk source code as individual methods. A strange, and key, feature of Smalltalk environments is that the programmer is editing a live system – accepted changes to source code affect the running environment. In order to work on separate projects, early systems required

<table>
<thead>
<tr>
<th>Feature</th>
<th>Sublime</th>
<th>Vim</th>
<th>Emacs</th>
<th>VisualWorks</th>
<th>Eclipse</th>
<th>Visual Studio</th>
<th>XCode</th>
<th>Cloud9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code representation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Workspace</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Tabs</td>
<td>Y</td>
<td>?</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Tab Groups</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Splittable Code Area</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Splittable Editor</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Multi-window</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Info Scrollbar</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Mini-map</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td><strong>Code navigation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File Hierarchy</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Relative Navigation</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Indirect Relative</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Call Hierarchy</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Find/Replace</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Find in Files</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Bookmarks</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Class Inheritance</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Class Composition</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Call Graph</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Class Browser</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 5.1: A comparison of features provided by a representative set of environments. IntelliJ, which is not shown for space reasons, has similar features to Eclipse. MacVim supports tabs.
copying the entire environment. VisualWorks Smalltalk now provides a project capability that allows separate projects. Smalltalk does not support tabs nor splitting method editors, but does allow multiple class browser windows. Navigation is usually achieved by selecting the desired package, class, method group, and method from contextualised lists. The class can be selected from either a package list or from an class inheritance hierarchy. Call hierarchy relative navigation is supported by selecting either ‘Browse Senders’, ‘Browse Implementors’, or ‘Browse Method’ from a contextual (right-click) menu. The environment also support sophisticated search facilities.

5.3.4 Integrated development environments

Visual Studio [Mic97] is a popular tab-based IDE for the Windows operating system that supports both projects and saving workspace state. Code editors featuring info scrollbars are opened within tab groups that may be split either horizontally or vertically about the central code editors area. Editor tabs may also be dragged out of the main window as new windows. Visual Studio supports indirect relative navigation by using its ‘go to definition’ functionality, as well as a call hierarchy view. The environment provides extensive find and find in files functionality, as well as bookmarking capability. Several auxiliary views are available including a file-browser view (Solution Explorer) and class composition view (Class View).

Eclipse [Fou01] is a one of the most popular IDEs for the Java programming language, and provides support for both projects and saving workspace state information. It provides extensive code editor functionality including tabs, tag groups, a splittable code area, splittable editors, multiple-window support, and an info scrollbar. It supports file/package navigation using several slightly different auxiliary views. It supports both both direct and indirect relative navigation, and also provides a call hierarchy view. It provides extensive search capabilities including find, and the equivalent to find in files, as well as supporting bookmarks. While, Eclipse shows a class inheritance tree using the ‘Type Hierarchy’ view if it is opened on the root Object class, it does not provide a class composition view, or call graph view. However, Eclipse does provide a sophisticated plug-in system so such views could be provided by third parties. IntelliJ [Jet01] is another popular Java IDE that provides comparable source code representation and navigation features to Eclipse except that it provides fewer auxiliary views and does not include a view that can show the inheritance hierarchy. As IntelliJ also provides a plug-in mechanism, such views may be provided by third parties.

Xcode [App03] is the premier IDE for software developed to run on Apple products. While many of its features are intended to support development in the Objective-C and Swift programming languages, the editor also supports syntax-highlighting for numerous other programming languages. The environment supports both projects, saving workspace state, and the use of multiple windows. In contrast to most other environments, while Xcode supports tabs, creating another tab replicates the entire workspace window not just the code area. The code editor
area itself may be split either horizontally or vertically by adding assistant editors, which may be set to automatically change based on the file in the primary editor, for example, to show the header file for an Objective-C source code file. Each editor has an info scrollbar similar to previously described environments. File-based direct navigation may be performed using the ‘Project navigator’ auxiliary view, and Xcode also supports both direct and indirect relative navigation, as well as navigation by a call hierarchy view. Xcode does not provide explicit support for bookmarks but does allow special comments to act as landmarks. Also, like Eclipse and IntelliJ, Xcode does not provide class inheritance or class composition tree graphs.

5.3.5 Web-based environments

As a full review of online development environments is beyond the scope of this chapter, just one environment is discussed that appears to be the most feature complete of such environments. Cloud9 [Clo, HPH14] is an online development and collaboration environment that provides the user with a web-based code editor that manipulates resources located on a containerised operating system. The interface provides a traditional tab-based editing environment that features syntax-highlighting and code completion. The interface is reminiscent of Eclipse with a tree-based file-browser on the left, an editors area in the middle, a tools area on the right, and a console area at the bottom of the screen. The editor area may be split either or both, horizontally and vertically, allowing at most four editors to shown at once. Like Eclipse, relative navigation may be performed by holding a modifier key and hovering the mouse over a reference to a declaration. The code editor part of Cloud9 is available separately as the Ajax.org Cloud9 Editor (ACE) [Pro06], and is used to provide code editor functionality in numerous other projects – for more information refer to the ACE Wikipedia page [Wika].

5.4 Summary

Software development environments have steadily improved as underlying hardware has increased in performance, and computer displays have increased in size and resolution. Early environments first focused on preventing the programmer from making syntax errors, then sought to provide better support during debugging by displaying runtime information. Only recently have prototype development environments (CodeCanvas and CodeBubbles) been developed that attempt to challenge the file editor paradigm that is pervasive in mainstream development environments – these environments provide a desktop metaphor that aims to support implicit and direct navigation by improving support for spatial memory. In contrast, tools that support relative navigation have only been implemented as auxiliary views via plug-ins to existing environments. The most advanced mainstream development environments provide similar features such as tabs, splittable tab groups, and multiple window support. They also provide varying degrees of support for relative navigation from contextual menus that provide
‘go to definition’ functionality to hyperlink-enabling method calls (and other references) to allow instantaneous navigation to the target definition – however, an issue with such relative navigation is that it often obscures the existing programming context.
Part II

Theory

The previous part provided a review of background literature in four important areas. From Chapter 2, we find that mental models exist on a continuum between a direct reflection of the real world through to very abstract models used for reasoning. From Chapter 3, we find that although extensive research has investigated the cognitive processes of programmers during program comprehension, other types of mental model, such as mental imagery, have been investigated to a far lesser degree – only recently has the concept of codespace been proposed that links program comprehension with spatial memory. From Chapter 4, we find that visualisation of software has been extensively investigated, and while some tools have sought to leverage spatial memory, such tools still allow the existing programming context to be obscured during navigation negatively affecting spatial memory. From Chapter 5, we find that several recent prototype development environments provide explicit support for either direct or implicit navigation, but do not aim to provide explicit support for relative navigation.

The first chapter of this part – Chapter 6 – describes how software is structured, discusses the types of computer systems used by programmers, and discusses programmer activities. The remaining chapters of this part gather together the three main theoretical contributions of this thesis. Chapter 7 proposes the attributes that must be present in visuo-spatial programming interfaces, and describes several existing systems that are considered examples of visuo-spatial programming environments. Chapter 8 describes the method-flow visualisation technique, which is a way of ensuring editors remain spatially consistent on a constrained screen, and may be used to navigate software within a scrollable flow view that leverages visuo-spatial memory by increasing visual momentum. Chapter 9 proposes the Software Dimensions concept, which refines the notion of layers of abstraction to take account for the difference in abstraction between composition and inheritance, and informs how best to display software source code on a two-dimensional screen, and specifically, within a method-flow column editor – to mitigate the hyperspace problem.
Chapter 6

About programming

Programming activities involve creating, navigating, comprehending, and modifying source code files. Once a software project has grown beyond a non-trivial size, displaying and navigating its source code becomes increasingly problematic. This chapter provides an overview of some issues that affect programmers. Section 6.1 provides a primer on how source code is structured and explains how source code is distributed amongst multiple source code files. Section 6.2 describes the different types of computer system that programmers use. Section 6.3 describes the different types of identifiable programming activities.

6.1 The elements of software

Typically, software is produced in a textual form within text-based source code files. While some development environments have stored source code using a binary, non-human-readable file format, most programmers prefer text-based development systems as non-human readable data-structures are susceptible to data corruption, and cannot be used with third party tools that expect text-based source code files. Modern IDEs now use a hybrid approach where source code is parsed into an internal representation that is later saved back to the original textual representation if changes are made.

Source code files contain definitions and declarations, which are sections of text that follow the rules of a programming language’s syntax, and have semantic meaning. The most common of these are global variables, data-structures, procedures, and functions. Object-oriented programming languages also allow the definition of classes, which associate a data-structure with methods – procedures that are bound to the class. Conceptually, any definition may be referred to by any other – for example a method calling another method (or itself), or referring to a global constant, but, in practise, a programming language provides scopes that restrict what code is allowed to refer to a definition, or hide them so that they can only be accessed by explicitly naming the scope. Such scopes are often referred to as namespaces, or packages.
Figure 6.1a shows the C++ source code for a program that adds the values of command-line arguments and outputs the resulting sum. The source code contains two top-level definitions: the `main` function; and the `math` namespace. Within the `math` namespace, the `sum` function is said to be defined within the scope of the `math` namespace. To reference the `sum` function, the `main` function must prefix the method call with `math::`. The C++ programming language was chosen for this example because it allows definitions to be mixed within the same source file, and allows a function to be defined within an explicit scope. The emergent structure of the source code can be represented as a call graph diagram, which allows the programmer to view a higher-level representation of the software (see Figure 6.1b).

```cpp
#include <stdlib.h>
#include <stdio.h>

namespace math {
    int sum( int nr, char** numbers )
    {
        int sum = 0;
        for ( int i=0; i < nr; i++ )
        {
            sum += atoi( numbers[i] );
        }
        return sum;
    }
}

int main( int argc, char** argv )
{
    int sum = math::sum( argc, argv );
    printf( "%\n", sum );
}
```

(a) The source code of the program

(b) The call graph of the program

Figure 6.1: A C++ program that sums integer arguments
Figure 6.2 shows the C source code for a program that calculates the area of a rectangle and adds it to the area of a circle. Within the source file there are three constants, three structures, and three functions. The rectangle and circle structures store data associated with those shapes and both also include the shape structure, which records the colour of the shape. Structures allow composite types to be created, which can be declared and passed like primitive types. A problem with this style of programming is that there is no rule for how definitions are grouped. Should the \texttt{area\_of(struct Rectangle \}) function be declared in the same source code file as the \texttt{Rectangle} structure, or with the \texttt{area\_of(struct Circle \}) function?

```c
#include <stdio.h>
#include <math.h>

const char* red = "#FF0000";
const char* green = "#00FF00";
const char* blue = "#0000FF";

struct Shape
{
    const char* colour;
};

struct Rectangle
{
    struct Shape shape;
    int width;
    int height;
};

struct Circle
{
    struct Shape shape;
    int radius;
};

float area\_of(struct Rectangle shape )
{
    return shape.width * shape.height;
}

float area\_of(struct Circle shape )
{
    return M\_PI * shape.radius * shape.radius;
}

int main( int argc, char** argv )
{
    struct Rectangle shape1 = { blue, 20, 40 };
    struct Circle shape2 = { red, 100 };

    float area = area\_of( shape1 ) + area\_of( shape2 );
    printf( "Area: %fn", area );
}
```

Figure 6.2: A C program that calculates the summed area of a rectangle and circle
Figure 6.3 shows the source code for a program – similar in function to that of Figure 6.2 – implemented using the Java object-oriented programming language. Rather than include shape properties through composition, the Rectangle and Circle classes inherit from the Shape class meaning that they inherit any members or methods defined the parent class. Additionally, now within the main method both the Rectangle and Circle objects can be treated as Shape objects.

```java
import shapes.*;

public class Main {
    static String red = "#FF0000";
    static String green = "#00FF00";
    static String blue = "#0000FF";

    public static void main( String[] args ) {
        Shape shape1 =
            new Rectangle( blue, 20, 40 );
        Shape shape2 =
            new Circle( red, 100 );
        double area =
            shape1.areaOf() + shape2.areaOf();

        System.out.println( "Area: " + area );
    }
}

package shapes;

public abstract class Shape {
    public String colour;

    public Shape( String col ) {
        colour = col;
    }

    public abstract double areaOf();
}

package shapes;

public class Rectangle extends Shape {
    int width;
    int height;

    public Rectangle( String col, int w, int h ) {
        super( col );
        width = w;
        height = h;
    }

    public double areaOf() {
        return width * height;
    }
}

package shapes;

public class Circle extends Shape {
    int radius;

    public Circle( String colour, int radius ) {
        super( colour );
        radius = radius;
    }

    public double areaOf() {
        return Math.PI * radius * radius;
    }
}
```

Figure 6.3: A Java program that calculates the summed area of a rectangle and circle.
For all but the most trivial software projects, source code is distributed among multiple source code files. While object-oriented programming enforces a more structured approach that allows better reasoning in terms of objects, unfortunately, it can also dramatically increase the number of individual source code files within a codebase, which, increases the need to navigate between the files during software exploration. As can be seen in Figure 6.3, the single source code file of Figure 6.2 produced four separate source code files when implemented in Java. While Java does allow nested classes, such functionality should only be used if it makes sense to treat a class as a subordinate of another. For example, in Figure 6.3, should the Rectangle and Circle classes be nested within the Shape class? In the future, if a user created their own shape, such as a Triangle class, by extending the Shape class, this might cause confusion as a rectangle would be instantiated by calling the Shape.Rectangle constructor, while a triangle would be instantiated by calling the Triangle constructor.

To illustrate the number of source code files that might exist in a typical project, and emphasise the difficulty of displaying a meaningful subset of a program’s source code on screen, Figure 6.4 shows the class composition hierarchy of the Visuocode prototype development environment, which was developed as part of this project. Each class represented in the figure by a box contains considerably more methods (and therefore lines of code) than the classes presented in Figure 6.3. Also, Figure 6.4 doesn’t show transient objects that are instantiated and destroyed within methods.

In Figure 6.4, if a class type is used in multiple places within Visuocode, it is represented in

Figure 6.4: For illustrative purposes: the Visuocode class composition model

Greyed classes are duplicated elsewhere in the diagram
the figure as a greyed class – for example, \textit{ClassSignature}. This illustrates another important issue when navigating source code. As all instances of a class are based on the same class source code file, if the programmer navigates to such a class, they are unlikely to remember all other locations where that class type is referenced. If source code changes are made to accommodate one area of the codebase, those changes may also affect execution in other areas. Therefore a programmer often wants to be able to determine which methods call the current method – these are referred to as \textit{called-from} links. Navigation back along such a link is referred to as a \textit{reverse navigation}, and such a navigation results in opening a method that is on a different branch of the call graph.

The valid syntax, and the semantic meaning, of source code declarations and definitions differ depending on the programming language they are written in. Most contemporary programming languages allow structures or classes to be dynamically allocated in memory, which allows complicated data structures to be formed, and then passed as arguments to functions and methods. Object-oriented languages, such as C++, Objective-C, Java, and C#, support class inheritance, which means that extending classes automatically include any members or methods present in ancestor classes; as well as \textit{interfaces}, which specify commonalities between different classes allowing them to be treated as a single type. Recent programming languages, such as Go [GPT07] and Rust [Hoa06], have re-imagined the concept of object-oriented programming to remove inheritance while retaining interfaces. Unfortunately, as programming languages support more structural semantics, it becomes harder for programmers to navigate source code written in those languages.

6.2 Computer systems used for programming

Displaying and navigating source code increases in difficulty as the amount of source code grows and the number of source code files increases. For any but the most trivial of software programs, it proves impossible to fit all of a program’s source code onto a single screen. Figure 6.5 shows a screenshot of an Eclipse environment that was taken on an 11 inch display – while four editors have been manually arranged within the environment to show as much code as possible, most programmers that use Eclipse would likely only have one editor open with the other files available in tabs. Arranging editors as shown in Figure 6.5 is both physically tedious, and proves frustrating as often the length of one or more lines of source code are too long to fit within the width of the editor’s pane.

Increasing screen size is not a solution. The large screen paradox [Atw07] states that the larger a monitor is the less useful it becomes to maximise a window on the screen due to the amount of unused space. Figure 6.6 illustrates this by showing the difference in screen area between the 11 inch laptop of Figure 6.5 and a 27 inch display that has a similar Eclipse window maximised (though with a right-hand \textit{outline view}). On the larger screen, each editor contains a vast amount of whitespace that is not being used, however, if another column of editors were
Figure 6.5: An Eclipse IDE window with source files manually arranged

Figure 6.6: Screenshots comparing Eclipse on an 11 inch laptop to a 27 inch display
added, the editors would be just as cramped as if using two columns on the smaller screen. Additionally, the interface would provide no spatial consistency between the six editors that would be visible.

Due to the difficulty of optimising screen real estate, the computer system preferences of programmers are often influenced by other factors, such as mobility. If a programmer can only easily view a section of one source file at a time, why use a desktop system and be bound to a desk? Also, modern laptops are easily able to drive the most demanding of external peripherals if needed. Conversely, other programmers prefer multi-monitor systems that allow specific monitors to be used for specific applications. Also, the type of project that a programmer is working on might influence the type of computer system they use. Often programmers that use multiple applications at once prefer to have a computer system that allows each application to be open on a separate screen. For example, during web development, one may want one screen for a visual design application, one screen for a software development environment, and another for a web browser.

6.3 Programming activities

Programming involves a number of distinct activities that each make different demands on the software development environment. These include: software navigation, which involves moving between different source code definitions by performing software navigations; program comprehension, which involves reading source code to understand the effects of execution; localised debugging, which involves analysing source code to locate where an unintentional effect is occurring; modification, which involves changing existing source code; and program composition, which involves creating new source code files or writing new software definitions. Programming usually involves swapping back and forth between these activities, and some tasks involve a combination of two or more of these activities – software understanding involves software navigation and program comprehension; debugging often involves software navigation and localised debugging; and modification is often proceeded by software navigation to find where the modification should be made. Software development also involves numerous other activities, such as design or testing, but they are not discussed here.

6.3.1 Environment support for software navigation

As previously stated, this thesis distinguishes between several different modes of navigation that are used to navigate software structure. It is re-emphasised that navigations are categorised based on how the programmer performed the navigation not why the programmer performed the navigation. Different development environments provide differing levels of support for each type of navigation. The following sections describe, for each navigation type, how environments may support that kind of navigation. These sections refer to capabilities previously described...
in Chapter 5 – refer to Section 5.2 and Section 5.3.

**Direct navigation**

Direct navigation occurs if a programmer navigates directly to a declaration using an auxiliary view. Most IDEs provide numerous auxiliary views that may be used to perform direct navigations, such as a file-browser view, class view, or an errors or problems view. As search views are both auxiliary views and allow the programmer to navigate directly to arbitrary locations in source code, navigations using search views are also considered direct navigations. In particular, programmers who use code editors such as Vim and Emacs, which do not provide graphical auxiliary views, tend to use search views to speed up opening source code files.

Some research environments have sought to provide extra support for direct navigation by providing a representation of a software codebase that is spatially consistent allowing the programmer to navigate by remembering a spatial location in the representation instead of a textual label. The Software Terrain Maps system presents a map of a codebase similar to a cartographic map that allows the programmer to navigate directly to a source code region, while the Code Thumbnails and Code Canvas systems provide thumbnail views of source code.

**Relative navigation**

Relative navigation occurs if the environment allows a programmer to navigate according to the emergent structure of the software. Different environments provide varying levels of support for relative navigation. Some environments, such as Eclipse and Xcode, are able to hyperlink-enable method calls and other references allowing the programmer to navigate directly to a definition, while other environments provide a ‘go to definition’ option in a contextual menu that, depending on the exact label selected, may take the programmer to a member declaration or the members class source code file. Several environments – including VisualWorks Smalltalk, Eclipse, Visual Studio, and Xcode – also provide a ‘Call Hierarchy’ view that can refreshed to show the methods calling and called by the current method. The Stacksplorer plug-in for Xcode provides additional support for relative navigation by indicating calling methods and methods called by the current method. Similarly, the Blaze plug-in provides additional support for relative navigation by providing a view that shows the current call graph branch. Code Bubbles allows called methods to be budded of adjacent to the calling method, which allows the programmer to navigate down a call graph branch to an arbitrary depth.

**Scrolling navigation**

Scrolling navigation occurs if a programmer scrolls through a file between method definitions or member declarations. Most modern IDEs augment the scroll bar with colour lines or boxes that represent information about locations in the source code file. For example, in Eclipse, a red box or line indicates the location of a detected syntactic error, and if a member or variable
is selected, the scroll bar shows where it is defined and other locations it is referenced. Code editors, such as Vim and Emacs, also provide advanced keyboard shortcuts that allow the programmer to navigate around a source code file very quickly – these are also considered scrolling navigations.

**Implicit navigation**

Implicit navigation occurs if a programmer moves their attention between definitions, or windows, with no explicit user input. Environments support implicit navigation by allowing the programmer to arrange and view multiple source code files at once. Several environments, such as Sublime, Vim, Emacs, IntelliJ, and Cloud9, allow the programmer to split code editors both horizontally and vertically allowing multiple editors to be shown. In particular, Vim and Emacs allow code editors to be continually split in half, either horizontally or vertically, allowing useful arrangements of editors. The Code Canvas and Code Bubbles environments allow methods or classes to be arranged within a desktop workspace so that a working set of files can be positioned together.

**6.4 Summary**

Any non-trivial software project contains more source code files than can be easily navigated using mainstream programming environments. Programmers often want to be able to easily follow the flow of execution between source code files that are adjacent on the call graph, as well as the inheritance and composition hierarchies. However, when using mainstream environments, such a navigation usually causes the existing programming context to be obscured by the next. To mitigate this, mainstream environments do provide features such as multi-window support, multiple, splittable tab groups within a window, and splittable editors, however, such features are quickly compromised by lack of screen real estate, and also practical human limits on the size and number of screens a programmer can comfortably use. To mitigate this problem, programmers often form a working set of source code files that are relevant to the immediate task, which are left open in tabs. Existing mainstream environments are not able to fully leverage visuo-spatial memory because they often have editors occupying the same spatial location, and provide no visual transition when the editor changes – a symptom of low visual momentum. This compromises the integration of navigations into a spatial cognitive map because any *programming artefacts* viewed have few spatial attributes. This, in turn, compromises the effectiveness of software navigation. To address these issues, the following chapters propose requirements for ensuring environments can leverage visuo-spatial memory, a visualisation technique to increase visual momentum within development environments, and a way to map the multi-dimensional aspects of software structure to a constrained set of three dimensions that can be represented on a 2D screen.
Chapter 7

Visuo-spatial programming interfaces

As humans are able to remember spatial information in a seemingly effortless manner, it has been natural to try and leverage spatial memory when designing the user interfaces of programming tools. I introduce the term ‘visuo-spatial environment’ to refer to software development environments that attempt to leverage spatial memory through the use of the visuo-spatial sketchpad component of working memory. Several existing development tools, including the traditional plain text code editor, may be considered visuo-spatial environments, however, even though a programming interface may leverage visuo-spatial memory, it may not do so in a way that supports the visualisation and navigation of software structure. Therefore support for visuo-spatial memory is not the end goal; the interface must also be designed to support programming.

While researching the literature related to human memory, as well as investigating development tools that claim to leverage spatial memory, several properties have been identified that appear necessary for such an environment to successfully leverage visuo-spatial memory. As short-term visuo-spatial memory is the conduit through which information is stored in long-term spatial memory, the following requirements should be relevant for any environment claiming to leverage spatial memory. As previously noted, the tools implemented during this project – Code-flow and Visuocode – are only considered examples of visuo-spatial environments. Other tools may have drastically different user interfaces and yet be classed as visuo-spatial environments.

Section 7.1 reviews everyday understanding of spatial memory and provides a thought experiment that justifies the required properties for visuo-spatial environments. Section 7.2 discusses each requirement. Section 7.3 discusses design decisions that must be made when designing a visuo-spatial programming interface. Section 7.4 describes existing environments that are considered visuo-spatial programming environments, and why.
7.1 Spatial memory

Before discussing the properties that visuo-spatial programming interfaces should have, spatial memory is first reviewed. Firstly, it is important to consider that while cognitive science researchers have investigated the capacities of the human memory system (reviewed in Chapter 2), understanding of the capabilities of spatial memory is based on collective personal experience. So far, controlled experiments have only attempted to measure aspects of the spatial system related to learning – for example, remembering artificial grid images after a brief exposure, remembering the location that a point of light was placed after a brief exposure, and remembering a sequence of blocks that have been tapped (see [CK16] for a recent review). In the field of navigation, more realistic experiments have been carried out that have individuals navigate through areas, however, these are also designed to investigate learning [CLS16].

Spatial memory provides humans (and presumably a lot of other creatures) the ability to remember the world that can no longer be seen. Spatial memory is able to store features that are to some degree fixed such as buildings, roads, rivers, oceans, etc; as well as features that may move such as cars, buses, planes, boats, people, animals, etc. Spatial memory is linked with episodic memory as locations may be remembered by thinking of past experiences at those locations, and similarly past experiences can be recalled by thinking about locations. The representation of such spatial information within long-term memory is referred to as a cognitive map [Tol48].

Spatial information can be described as being either egocentric - relative to the person; or allocentric - relative to the environment [BMO02]. Objects within our immediate environment are usually considered relative to ourselves – for example, my coffee cup is in front of me, my kitchen is behind me. Alternatively, we are able to mentally consider ourselves from outside our own body and imagine our current location relative to other locations.

Most people are reliant on their visual facilities to orient themselves – to mentally understand where in their cognitive map of the world they are. This is achieved through one’s sense of sight, one receives a 2D image on the retina of each eye that is processed into a 3D representation within the visuo-spatial sketchpad [Bad12]. Through as yet unknown processes this representation is able to activate long-term memory and make memories available to consciousness, or form new memories [Log14]. For example, when a child wakes up and finds themselves in bed, they are able to orient themselves even though they fell asleep in the car the night before. The child is able to orient themself by recognising the shape of the ceiling above them, the feel of the bed they are in, the smell of the room, and other sensory information.

It is important to recognise that the visual stimuli being processed by visuo-spatial memory is based on where a person is looking, and therefore is constantly changing. However, the relative positions of objects within a person’s visual field remain spatially consistent, as do the spatial positions of objects person can no longer see. On a computer screen, this can be simulated for a 3D visual by panning the viewpoint. Conceptually, a window scrolling is similar,
just in 2D, which is no doubt why scrolling is such a popular metaphor in modern computing.

Further appreciation of the capabilities of spatial memory requires a thought experiment. Consider you are staying in a hotel that has many identical suites. If you were blindfolded and taken to a suite, upon having the blindfold removed, how would you be able to determine if you are in your own suite or that of someone else? First, you might look to see if your own personal items are present? Possibly, you may have left a watch by the bed. Is there a watch by the bed? Does it look the same as yours? Possibly, you may have left sunglasses on the table. Are there sunglasses on the table. Do they look like yours? Only if the possessions you expect to see are present will you be able to orient yourself as being in your own suite. However, what if you see your sunglasses on the table and your watch by the bed, but they are both broken? In this situation, even though the items are not exactly as you remember them, you may still orient yourself within your own suite because for both personal items a recognised change has occurred. This scenario relates to the first property for visuo-spatial programming interfaces – such environments must ensure that programming artefacts remain visually distinct.

What if your possessions are present, but you had left them in a heap on the bed? Would you be able to remember exactly how you left them? Chances are you would not. Recent research indicates that the capacity of spatial memory is not a fixed number of slots, but actually corresponds to the distribution of visual objects around the visual cortex – spatially separating objects increases the number of objects that can be recalled, while overlapped objects are prone to being lost [FAC13]. This scenario relates to the second property for visuo-spatial programming interfaces – such environments must ensure that programming artefacts are spatially separated.

What if your possessions are present, but not in the locations you left them? There is also the possibility that whoever blindfolded you also retrieved your possessions and placed them about the room in a similar, but not identical, arrangement to how they were before. In this situation, you may orient yourself as being in your own suite, or you may decide that you are in another suite. Only if your possessions are exactly where you left them would you be able to orient yourself as being in your own suite. This scenario relates to the third property for visuo-spatial programming interfaces – such environments must ensure that programming artefacts are maintained within a consistent environment that preserves spatial placement.

However, there is also a temporal aspect to spatial memory. If you have been staying in the hotel room for several days it may be the case that you have left your possessions in various different locations within your suite on each day. Now that you have had the blindfold taken off, can you remember where you last left your possessions in the suite? If you always left your possessions in the exact same places every day, you would be able to remember exactly where they should be. If, however, you vary the positions of your possessions, it will be likely that you will only be able remember the locations you left them most recently; and if several days have past since you were last in the room, it is unlikely you will be able to remember the locations of the items at all. This scenario relates to the fourth property for visuo-spatial interfaces – an
interface must maintain spatial consistency for an appropriate amount of time. The length of
time that is appropriate depends on the purpose of the interface.

Finally, if your hotel suite has a window with a view of a landmark you recognise, you will
be able to easily orient yourself within your own suite. This scenario relates to an optional
requirement for visuo-spatial interfaces – such environments should be developed to provide
landmarks that may help to orient the programmer.

In the next section, these requirements are discussed in greater depth in the context of
programming.

7.2 Requirements for visuo-spatial interfaces

In a similar way to the thought experiment above, when programmers are presented with
source code, the human memory system needs to recognise the visuo-spatial characteristics
of the source code that they see in order to properly orient themselves within the codebase.
A visuo-spatial programming interface should be designed to support such orientation. It is
important to recognise that the cognitive maps that are formed from software may persist for
varying lengths of time. One of the most important choices that a designer makes for their
interface is how long they expect the cognitive map to persist for. This is discussed further in
Section 7.2.4

A programming artefact is analogous to a sequence of characters from a source code file that
may be arbitrarily reordered with other similar artefacts. For example, a method is considered
a programming artefact as it may be moved above or below other methods without affecting the
execution of the software; and a class is considered an artefact if the programming language
supports multiple classes within the same source code file, or supports inner/child classes.
Similarly, enumeration and member declarations are considered programming artefacts.

A visuo-spatial programming interface must:

1. Ensure programming artefacts remain visually distinct.
2. Maintain the spatial separation of programming artefacts.
3. Maintain a spatially consistent environment.
4. Maintain spatial consistency for an appropriate period of time.

Formally: A visuo-spatial programming interface maintains the spatial separation of visible
visually distinctive programming artefacts within a spatially consistent environment possibly
associated with a programming landmark for some period of time.
7.2.1 Visual distinctiveness of programming artefacts

Visuo-spatial environments must maintain the visual distinctiveness of programming artefacts so that they act as memory cues during software navigation. If the appearance of programmatic artefacts change, they may no longer be able to cue where the current location corresponds to in the cognitive map. Therefore it is important that the visual appearance of a method remains unchanged each time it is viewed. An exception to this is if the change can be recognised as an alteration of the previous state. This has ramifications for environments that automatically wrap long lines and in so doing alter the appearance of methods. This is supported by recent research investigating how much detail is stored in long-term memory when viewing images – participants had an 87% success rate at detecting which of two very similar images they had previously seen [BKA08], which implies that programmers are able to distinguish between different methods visually, and that a small change to a method’s visual representation may affect its recognisability, unless it is an expected or recognisable change.

7.2.2 Spatial separation of programming artefacts

A visuo-spatial programming interface must provide each programming artefact a distinct spatial location that does not overlap any other artefact. Recent research indicates that the capacity of spatial memory is not a fixed number of slots, but actually corresponds to the distribution of visual objects around the visual cortex – spatially separating objects increases the number of objects that can be recalled, while overlapped objects are prone to being lost [FAC13]. It follows that spatially separated programming artefacts are better integrated into a representation within long-term memory than if artefacts overlap or obscure each other. It is emphasised that this requirement does not insist that the position of artefacts is fixed, only that programming artefacts do not overlap each other. Traditional environments display source code files in tabbed editors that overlay one another, which interferes with spatial memory because all source code files are spatially in the same location. While this can be mitigated if using an environment that supports tab groups or splittable editors, only a limited number of code editors are able to be displayed in this manner, and they must be manually arranged. Therefore, a visuo-spatial programming interface needs to provide a mechanism that prevents methods from being placed on top of one another, but that also allows methods to be placed near related code in an intuitive position.

This requirement is controversial because it immediately requires that a visuo-spatial programming interface use a paradigm that is different to that of the traditional tabbed code editor. The most obvious way of meeting this requirement is to use a desktop metaphor where programming artefacts are positioned spatially on a desktop and the system prevents artefacts from overlapping, however, any interface that prevents artefacts from overlapping (such as a scrolling window) would meet this requirement. It has been argued that the spatial separation of editor tabs may be enough to spatially identify programming artefacts. The problem with
this is that the programming artefact is only recognisable from the name of the source file. During the studies carried out as part of this project, programmers were often observed to click on tabs only to find that they had selected a different file to the one they were expecting.

7.2.3 Ensuring a spatially consistent environment

A visuo-spatial programming interface must also ensure that the spatial relationships between visible programming artefacts are maintained. Even if an environment consistently separates programming artefacts, visuo-spatial memory will not be supported if their locations are jumbled each time a new method is added.

In everyday life, in order to orient ourselves, we rely on the spatial position of certain things to remain the same. Similarly, in order for a programmer to orient themselves within their cognitive map of a piece of software, it is important that the relative spatial relationships between programming artefacts remain unchanged. It has been also argued that the spatial position of editor tabs may be enough to maintain a consistent environment for programming artefacts. In addition to the tabs only being recognisable by file names as previously discussed, another problem is that the arrangement is only valid for the length of time that those editors tabs are positioned in the same way. This issue relates to the fourth property of a visuo-spatial programming environment – the environment must be designed to maintain spatial consistency for an appropriate amount of time. If the visuo-spatial interface designer has decided that it is acceptable that consistency is only maintained for the lifetime of a unique arrangement of tabs, then that interface would be described as having ‘task consistency’. This is discussed further in the next section.

It is important to note that spatial consistency need only be maintained between visible programming artefacts. In everyday life, it is easier to keep track of the spatial position of objects if they are within an enclosing context – such as a room. A visuo-spatial programming interface might also be developed that uses a similar analogy.

It is emphasised that the reason for designing a visuo-spatial interface is to leverage long-term spatial memory, which primarily stores the spatial location of objects that can no longer be seen within a cognitive map. Therefore, the main purpose of a visuo-spatial programming interface is to support the integration of spatial relationships between programming artefacts into long-term memory. Again, the most obvious way of meeting this requirement is to use a desktop metaphor that maintains the spatial position of programming artefacts, but that is able to scroll so that the programmer can focus on only those artefacts that are necessary. Again, this may also be implemented within a scrolling window that maintains a consistent order of programming artefacts that is appropriate for the intended persistence of the programmer’s cognitive map.
7.2.4 Temporality

The rationale for providing a visuo-spatial programming interface is to leverage spatial memory by ensuring that an artefact is consistently spatially positioned for long enough to be both stored and recalled from long-term spatial memory in a meaningful way. As human spatial memory is able to effectively handle objects that maintain spatial positions for varying lengths of time, different visuo-spatial interfaces may be designed that maintain the spatial positions of programming artefacts for differing lengths of time. The temporality of an interface – how long a cognitive map is expected to persist – is likely the most important design choice for a visuo-spatial interface as it becomes a constraint for other decisions. I suggest that the duration of consistency should be described as either: permanent, persistent, sessional, or task. Permanent consistency means that the spatial arrangement is generated based on the inherent structure of the software. Persistent consistency means that the spatial arrangement is persisted over multiple sessions until discarded by the user. Sessional consistency means that the spatial arrangement only lasts during a programming session. Task consistency means that the spatial arrangement only lasts for the duration of a particular task.

7.2.5 Programming landmarks

Visuo-spatial programming environments should also attempt to provide programming landmarks that support navigation relative to each landmark. Landmarks are often used to aid real world navigation by providing a point that local navigation can then proceed in relation to. Research suggests that humans build up a hierarchical structure of landmarks, which they then use to support navigation [MHH89]. It has been previously proposed that, in software, comments and groupings of methods also represents a style of landmark [CFO05]. How a system might represent landmarks depends upon the style of interface that is provided. Potentially, landmarks may be represented by colouring the background of certain editors to indicate that they belong in a certain package or group; alternatively, editors might indicate the navigation path that has been followed to reach the current location.

7.3 Design decisions for visuo-spatial interfaces

Designers of visuo-spatial programming environments need to address several issues related to the positioning of programming artefacts within the environment. Real world environments that are remembered within spatial memory have a spatial arrangement that, though potentially temporary, is canonical for the duration of its existence as no object can occupy two different locations at once and no two objects can occupy the same location in space. In contrast, visuo-spatial programming interfaces may provide multiple different spatial arrangements of the same programming artefacts. A possible reason why text editors have remained so popular is that
they do provide a canonical spatial representation of the programming artefacts contained within them. The following is a list of some design decisions that have been identified as needing to be decided when designing a visuo-spatial programming interface:

- Should the designer choose an unstructured or structured interface?
- How should the system choose the codebase subset to represent?
- How long should spatial consistency be maintained for?

### 7.3.1 Unstructured or structured interface?

A crucial decision is whether the user interface is unstructured or structured. An interface that uses the desktop metaphor is considered to be an unstructured interface because the user may position artefacts anywhere on the desktop. Like the physical object the metaphor reflects, such an interface might allow multiple different types of artefact to be represented on the desktop. A limitation of the desktop metaphor, however, is that only a small number of artefacts (relative to the size of a codebase) can be represented without the risk of objects overlapping. Conversely, if only a small number of artefacts are represented then where have all other artefacts gone? Currently, several existing systems that use the desktop metaphor mitigate this by providing a scrollable desktop that allows multiple groups of programming artefacts. However, a key issue with such a representation is that any cognitive map formed is likely to represent only those artefacts represented and is only relevant to that arrangement of artefacts – this issue can be addressed to a degree by allowing a desktop to be algorithmically populated with a subset of the codebase so that spatial consistency persists for working sets, at least. While unstructured desktops can easily meet the requirement that programming artefacts are spatially separated, due to the limited amount of source code that can be displayed on a desktop, it is much more difficult for such environments to maintain spatial consistency. In contrast, a structured interface enforces the placement of programming artefacts. Mainstream environments such as Eclipse are considered to have a structured interface because code editors are opened within the active tab group. Structured interfaces have the problem that code editors are often opened in the same location, or one of a few locations (if multiple windows or tab groups are being used). The method-flow visualisation technique represents a structured interface because the system decides where to place an opened editor column – if a direct navigation is performed, the flow view is emptied and and the new editor column is placed within it; whereas if a relative navigation is performed, a new editor column is placed to the right, and adjacent to, the existing editor column.

### 7.3.2 How to choose codebase subset?

Due to the physical size constraints of monitor screens, as well as the practical constraints related to distance and viewing angle that appear with large monitors, only a subset of the
codebase of a piece of software can be displayed at once. Therefore, a design decision is how a subset of the codebase is chosen for representation. Interfaces that use the desktop metaphor either allow the user to spatially place the programming artefacts they want or allow a set of programming artefacts to be automatically chosen by the system. In contrast, a structured interface requires a rationale for what programming artefacts are visible and for how long. For example, the method-flow visualisation technique enforces the constraint that any editor columns within the flow view represent a path through the software’s emergent structure.

7.3.3 How long is spatial consistency maintained?

A crucial decision is how long the spatial consistency of artefacts is maintained, which determines how long a programmer’s cognitive map should persist. An unstructured environment that uses a desktop metaphor is only consistent for the length of time that the position of programming artefacts is maintained relative to one another. Potentially, if such an environment does not support scrolling the desktop, programming artefacts may need to be manually placed every time the programmer changes tasks – such an environment is described as having task consistency. However, an unstructured environment might provide a scrolling desktop that allows multiple groups of programming artefacts to be formed – such an environment is described as having sessional consistency. An unstructured environment may also persist the spatial locations of programming artefacts between sessions – such an environment is described as having persistent consistency. It is also possible that a environment is developed that uses a desktop metaphor that does not allow the programmer to position programming artefacts; instead programming artefacts are positioned algorithmically – such an environment is described as having permanent consistency.

7.4 Existing visuo-spatial programming environments

Several existing development tools, including the traditional plain text code editor, may be considered visuo-spatial environments. This section discusses several existing environments that are considered to be examples of visuo-spatial programming interfaces, and compares them to the definition provided above.

7.4.1 The text editor

The most well-known, programming tool that arguably contains a spatial interface is the plain text editor. Visually, a text editor represents a text file within a two-dimensional environment of whitespace that can contain characters at arbitrary locations. By convention, however, text editors maintain a number of sequential lines from the top of the environment, each of which is populated from either the left or right side of the environment by an arbitrary number of characters. Some text editors allow the user to split an editor into two independent views of
the same file while ensuring that both editor halves remain consistent. Within a text editor, programming artefacts are maintained in a consistent order, and each is represented as one or more sequential lines of text, which maintains their visual distinctiveness. Also, programmers are able to scroll through source files having remembered spatially where software artefacts are. Often, additional landmarks are added to source files in the form of comments to provide an additional level of navigation hierarchy. Text editors provide permanent consistency.

Even though a text editor can be considered to leverage visuo-spatial memory effectively, this support does not extend to navigation of other source code files. Also, within a source code file, programming artefacts that are adjacent on the call graph are unlikely to be adjacent within the source code file. Therefore while text editors are considered an example of a visuo-spatial programming interface, they do not support effective visuo-spatial programming.

7.4.2 Code Thumbnails

DeLine et al. have described their Code Thumbnails environment [DCM+06], which uses a Multiple Document Interface (MDI) to present source code file thumbnails within a Code Thumbnail Desktop. Each thumbnail shows the source code of a file at a font size that is just below that of readability so the “developer can use the text shape for visual landmarks” [DCM+06, p. 1]. The programmer may then open a specific location of a file within a code

Figure 7.1: Code Thumbnails [DCM+06]
From the visuo-spatial interface perspective, the programming artefacts are the source file thumbnails, which are visually distinctive, spatially separated, and are consistently spatially arranged as placed by the user, and the Code Thumbnail Desktop is the environment. The names of each source file can be used as contextual landmarks, as well as the rendered representation of the text within source code files. If Code Thumbnails supports saving the position of each thumbnail, it would provide persistent consistency, otherwise it would provide sessional consistency.

While Code Thumbnails allows the programmer to see the whole source codebase and allows them to navigate by remembering where a declaration is located on the Code Thumbnail Desktop, this is based on the software’s file-based structure rather than its emergent structure. A potential problem with the Code Thumbnails paradigm is that it provides little support for building up a hierarchy of landmarks – the programmer either sees the entire project desktop or a single source code file. In summary, Code Thumbnails provides an elegant mechanism for direct navigation, but does not provide any additional support for relative navigation.

### 7.4.3 Code Canvas

DeLine and Rowan later described the Code Canvas system which uses an infinitely zoomable surface [DR10]. Like Code Thumbnails, Code Canvas allows the user to rearrange source files on the surface, however, the user is also able to tear off individual methods within files, and spatially rearrange them within a containing box representing that file. Unlike Code Thumbnails, to edit individual files, or methods, the user is able to zoom in and out from methods through the use of a ‘semantic zoom’ meaning that all actions are spatially coherent.

The user is also able to create multiple surfaces that may be either positioned on other displays or presented as multiple tabs. Colour is used effectively to highlight groups of artefacts.

From the visuo-spatial interface perspective, Code Canvas provides a scrollable and zoomable surface which is the environment. While the visual distinctiveness of programming artefacts is maintained, the use of semantic zoom means that each artefact has more than one representation, which may cause some confusion. Within the environment, Code Canvas allows a hierarchy of spatially separated programming artefacts – directories contain source code files that contain methods. At each level, artefacts can be spatially arranged within their parent container, providing spatial consistency. If Code Canvas supports saving the position of programming artefacts, it would provide persistent consistency, otherwise it would provide sessional consistency.

Like Code Thumbnails, Code Canvas provides an elegant mechanism for direct navigation, with the addition of intermediate landmarks in the form of directories, and the ability to spatially rearrange methods. From the visuo-spatial programming perspective, a potential concern with Code Canvas is the ability to create multiple different surfaces that provide
different arrangements of the same source code. Like Code Thumbnails, it appears Code Canvas does not provide support for relative navigation apart from that due to manual arrangement of methods within files. However, Code Canvas does provide an execution visualisation that highlights the flow of execution using red lines and arrows, and also provides a view that collects together those methods that are currently on the call-stack into a single view.

### 7.4.4 Code Bubbles

Bragdon et al. have presented the Code Bubbles system, which displays individual methods as ‘bubbles’ on a large workspace. Instead of laying out all the source code of a project, Code Bubbles takes a working set approach that allows the programmer to display individual method bubbles [B+10, BRZ+10b, RBL12]. More methods may then be ‘budded-off’ to be displayed adjacent to a calling method – automatic layout ensures that no bubbles overlap. The intention
is that groups of bubbles are arranged along the workspace, which is able to pan to the left and right. Like Code Canvas, colour is used to highlight groups of methods.

From the visuo-spatial interface perspective, the bubbles provide a visually distinct representation of the programming artefacts that are spatially separated on the workspace, which is the environment. The ability to colour the background of a group of bubbles provides a landmark mechanism that may be used to navigate quickly between groups of bubbles. At the top of the screen, a bar that presents labelled representations of groups of bubbles may also help navigation. The spatial arrangement of Bubbles are able to persist over multiple programming sessions if desired, therefore it supports persistent consistency.

Code bubbles specifically supports implicit navigation by allowing the programmer to form working sets of program fragments within the workspace. In contrast to Code Thumbnails and Code Canvas, Code Bubbles does not provide any extra novel support for direct navigation beyond that of a traditional environment as the workspace is not intended to represent an entire source codebase. However, Code Bubbles does provide additional support for relative navigation by allowing called methods to be ‘budded-off’ and automatically positioned adjacent to the calling method on the workspace surface.

7.4.5 Patchworks

Recently, Henley and Fleming [HF14] presented their Patchworks system, which presents a 3x2 grid of editors called ‘patches’. Each patch can be used to edit code fragments such as
methods, classes, or files. They stress that a key decision was to “make the grid of patches fixed: the programmer can neither adjust the size of the patch nor the number of patches” [HF14, p. 2514]. The contents of each patch are determined by dragging a reference from an auxiliary file-browser, or by dragging the contents of a patch to another patch – resulting in the two patches swapping their contents. To mitigate the limitation of only seeing 6 patch editors at one time, Patchworks also provides a view that displays a “never-ending ribbon of patches” [HF14, p. 2514] called the ‘Ribbon’. The programmer is able to use this ribbon view to adjust what patches are visible.

From the visuo-spatial interface perspective, the patches provide a visually distinctive representation of the programming artefacts that are spatially separated within the Ribbon, which is the environment. It appears that arrangements of patches are spatially consistent during a programming session, and probably between sessions – providing sessional consistency.

A potential issue with the Patchworks system is the lack of landmarks to identify specific groupings of patches within the Ribbon. This could mean that in the longer term the programmer is unable to consistently remember the location and content of specific patches, and may be exacerbated by patch editors being scrolled, causing the programmer to no longer visually recognise those editors.

Like Code Bubbles, Patchworks provides extra support for implicit navigation between a working set of visible patch editors. From the information available, Patchworks does not provide any more explicit support for either direct navigation or relative navigation beyond that provided in mainstream development environments.

![Figure 7.4: Patchworks [HF14]](image-url)
7.5 Summary

In order to leverage visuo-spatial memory, it is proposed that visuo-spatial programming interfaces must spatially separate programming artefacts and ensure spatial consistency between visible programming elements relative to one another. Additionally, groups of artefacts should be able to be associated with some form of landmark to facilitate hierarchical ordering. Important design decisions for visuo-spatial interfaces include whether to use an unstructured or structured interface, how a codebase subset is chosen for representation, and how long the interface should maintain spatial consistency. The Code Thumbnails and Code Canvas systems are considered to be examples of visuo-spatial interfaces that provide explicit novel support for direct navigation as these systems display the entire codebase of a project. Code Bubbles and Patchworks are considered to examples of visuo-spatial interfaces that provide explicit novel support for implicit navigation as the aim of the systems is to enhance navigation between a working set of methods or files. However, none of these systems aim to provide explicit novel support for relative navigation, though Code Bubbles does provide additional support for relative navigation by allowing methods to be ‘budded-off’ from existing bubbles.
Chapter 8

Method-flow visualisation

The emergent structure of software results from method and function calls between source code files, as well as inheritance and composition relationships between data structures. The result is a non-linear structure composed of source code files linked by syntactic text that is similar to the structure of interconnected hypertext documents – such a structure is referred to as a hyperspace [Nie90, SFM97]. This chapter describes method-flow, which is a way of visually representing the navigation of software structure in such a way that increases visual momentum. Section 8.1 discusses issues related to the navigation and comprehension of hyperspace structures. Section 8.2 describes how such issues are currently mitigated. Section 8.3 introduces the flow view, which is a generalisation of method-flow for navigating hyperspace structures. Section 8.4 describes the method-flow visualisation technique.

8.1 Comprehension of hyperspace structures

Hypertext documents have been advocated as a medium to allow self-directed learners to browse through education materials, however, they have also been shown to led to disorientation as learners become ‘lost in hyperspace’ [San13]. The motivation for navigating hypertext structures is similar to the motivation for navigating software – in both cases the person is trying to integrate knowledge into a mental model. People reading and navigating hypertext documents experience many of the same issues that face programmers navigating software [SFM99], such as they must choose which links to follow in order to progress, performance is aided by prior domain knowledge, and they may become disoriented – failing to remember how they navigated to the web page they have open. Web browsers and software development environments share a common key problem – after a navigation action, the user’s existing context is replaced with a new context.
8.2 Current mitigation strategies

Web browsers and modern IDEs also share common strategies for mitigating disorientation. Both browsers and IDEs record the individual navigations that a person has made in a history that allows the user to press a back button to return to the previous web page, or source code file. However, if a user goes backward in their history, as soon as they perform a new forward navigation, they lose their existing forward history, which can be a problem as sometimes it can be difficult to remember which link was followed to arrive at a particular page. Both browsers and IDEs also provide tabs, within which pages or editors can be opened, however, systems can only show a limited number of tabs before the tab labels are obscured, and although the navigations a person makes form a tree-like structure, navigations are presented as a sequence of open tabs – if tabs aren’t closed when a person back-tracks, the sequence of open tabs does not reflect the user’s position within the navigation tree.

8.3 Flow views

As part of this project, the flow view has been developed to aid such situations. A flow view is a horizontally scrollable view that contains adjacent, application specific, view columns – similar conceptually to the column view in the Mac OS Finder. According to Wikipedia, such columns are originally attributed to Mark S. Miller of Yale University, and are referred to as both Miller columns and cascading lists [Wikb]. A web browser application might be implemented so that each view column contains a separate web page view, while an IDE might be developed so that each view column contains a text editor. During navigations, if a link is followed, any view columns to the right of the column containing the link – the active column – are removed and a new view column is added adjacent to the active column ensuring that the editor columns within the flow view always correspond to a valid navigation path. Otherwise, if view columns were not removed, the columns contained within the flow view would soon become jumbled,

Figure 8.1: A flow view is a horizontally scrollable view containing adjacent view columns
and therefore would not correspond to any existing cognitive map – potentially leading to disorientation. It is proposed that the use of a flow view increases the visual momentum provided by an interface.

8.4 The method-flow visualisation technique

The Method-flow visualisation technique has been developed as part of this project. A method-flow is a flow view that is intended for software development [BH13]. Formally: the method-flow software visualisation technique explicitly supports software navigation and composition by allowing the programmer to traverse a method call graph using adjacent editor columns within a scrollable flow view. In each editor column, method calls are presented as hyperlinks. If a hyperlink is followed, a new editor column is stacked immediately to the right of that column causing any existing columns to the right of that editor column to disappear. As more editor columns are added, the leftmost columns scroll off the left side of the screen.

The scrolling of editor columns from the screen is an integral part of method-flow as it maintains the consistent spatial orientation of the editors relative to one another. Although only a limited number of editor columns can be seen, the use of a virtual scrolling canvas allows an arbitrary number of editor columns to be retained and available if and when needed (see Figure 8.1). Because methods are not constrained by the width of the screen, method editors can be represented full width within a code editor without wrapping lines. This is important because it means that the visual representation of a method is not altered by lines being wrapped, which should better support visuo-spatial memory. While, it is conceded that this may lead to wide methods not fitting fully on the screen, it is believed that from the visuo-spatial memory perspective it is preferable if such methods are reformatted manually by the programmer so that they are presented consistently in the future. Further empirical investigation needs to be carried out to determine whether strategies used to intelligently wrap lines based on the syntax tree affect visuo-spatial memory differently to standard line wrapping or not wrapping lines.

For larger displays, about three or four methods can be displayed at once depending on the width of the methods and the width of the Flow window. For smaller displays – for example a laptop – about two or three methods can be displayed at once. Because the window content can be easily scrolled, editor columns can be easily brought back into view to allow the programmer to easily follow the flow of execution.

8.4.1 Benefits during comprehension, navigation, and composition

During navigation, the benefit of this technique is that a programmer can explore down a branch of the call graph without obscuring their initial programming context – at any time, the programmer is able to scroll back to the left to refresh their memory regarding what methods
they have traversed, or the behaviour of these methods. During program comprehension, the
benefit of method-flow is that methods are juxtaposed side-by-side, which is likely to reduce
incidents of thrashing between two different source code files. During program composition,
new code can be created within a new adjacent editor column. As the existing programming
context is maintained, the programmer is able to refer back to the calling method to ensure
that the code is consistent with the calling context.

8.4.2 Support for visuo-spatial programming
Method-flow is thought to provide support for visuo-spatial programming because methods are
presented in spatially distinct positions, allowing them to be encoded within the visuo-spatial
sketchpad of short-term memory without interfering with one another. This encoding is said
to be egocentric because it is relative the programmer’s current programming context. In turn,
the spatial information of methods traversed should then be automatically integrated into an
allocentric cognitive map within long-term memory. Theoretically, once integrated into such a
cognitive map, in the future, the programmer should be better able to ‘know’ how to navigate
to specific regions of the codebase using relative navigation.

8.4.3 Implementation specific concerns
The term ‘method-flow’ describes the technique of navigating through methods using editor
columns within a flow view. Apart from the requirement that method calls are displayed as
navigable hyperlinks, the method-flow visualisation technique does not specify what should be
displayed within each editor column. Potentially, the contents of each editor column may be
either a code editor that displays an entire source file, or it may be a more specialised view that
just shows the subset of a file. In the later case, the implementation also needs to provide some
mechanism for editing any aspects of the class that are not shown within the editors present in
the editor column – such as the class declaration or class members. Similarly, the method-flow
visualisation technique does not specify how the method of the leftmost column editor is chosen;
it is left to the implementation to supply a suitable mechanism for direct navigation.

It has been suggested that if the programmer backtracks to an earlier method in the flow
view, then navigates down another branch of the call graph, the system should remember any
methods that were navigated to in the earlier branch. This would require that the system
maintain a graph structure of navigations performed so that if the user starts navigating back
down the earlier branch, the system can pre-populate the flow view with methods. A potential
issue with such pre-population is what the system should do if the branches the user has
navigated down contain a common trunk. Should the system show the most recent branch, or
should the system show methods traversed to the first fork?

An important consideration is how an implementation handles virtual methods. Currently,
systems that support relative navigation open the source code file that corresponds to the
declared type of the called object, which often may be an abstract class or an interface, not the class of the method that is invoked at runtime. Potentially, an implementation could show a list of relevant types and let the user select the appropriate one.

Method-flow also does not specify whether an implementation should represent any methods that call the current method from other call graph branches – called-from links (back-links). While potentially a list of such links may be represented at the bottom (or within) a column view, the effect that following such a link might have on the flow view is undecided. Potentially, all of the editor columns to the left of the current method could be replaced with the selected method, however, this would require a ‘back’ feature that would allow the user to undo such a navigation. Alternatively, an extra flow view layer could be added in front of the existing flow view, which could be later closed to reveal the original flow view.

8.5 Summary

Flow views are a general visualisation technique that can be used for navigating hyperspace structures that increase visual momentum. Each column view within a flow view provides navigable hyperlinks that allow additional column views to be added to the flow view. A ‘method-flow’ flow view can be used for creating, visualising, and navigating, software structure. As methods remain spatially consistent within the flow view, method-flow provides support for visuo-spatial programming by allowing egocentric navigation of methods to be integrated into an allocentric cognitive map of the software codebase.
Chapter 9

The ‘Software Dimensions’ concept

Software is recognised as being a complex multi-dimensional artefact. The call graph, composition hierarchy, and inheritance hierarchy are important aspects to software structure, however, there are also other aspects to software that exist such as: the flow of objects that are passed as arguments to method calls, the relationship between the declaration of a member and its use within methods, and shared object instances. This chapter proposes how these multiple aspects that make up the structure of software can be mapped to a constrained number of dimensions that may be represented on a 2D screen – the ‘Software Dimensions’. The aim is to provide a representation that optimises the integration of software navigations into a consistent cognitive map.

If a web browser flow view were used to navigate hypertext, each hyperlink would open a new browser column because there is little semantic difference between hyperlinks. If such a hypertext document did not have a consistent hierarchy structure, readers may return to pages already visited negatively affecting the development of a cognitive map of the structure of the hypertext document. In contrast, the relationships that bring about the emergent structure of software do have semantic differences, such as method calls, composition, and inheritance, which can be used to map software structure to a flow view in a more intuitive and elegant manner.

As part of this project, the Software Dimensions concept has been developed where each further dimension represents a different class of semantic link. The concept was inspired by the following quote from Brad Myers who wrote:

“Computer programs, however, are presented in a one-dimensional textual form, not utilising the full power of the brain. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs, have long been known as helpful aids in program understanding.” [Mye86, p. 60]

This chapter describes how Software Dimensions may provide software with a canonical structure that allows it to be laid out in a consistent and intuitive manner on a 2D display. Section 9.1 explains the ‘Software Dimensions’ concept. Section 9.2 describes how the Software
Dimensions may be supported within a development environment. Section 9.3 discusses design decisions related to supporting the Software Dimensions. Section 9.4 discusses limitations of the Software Dimensions concept. Section 9.5 conjectures how code may be represented within a spatial cognitive map.

9.1 Mapping to three ‘Software Dimensions’

The term ‘layers of abstraction’ is widely used within computing, but arguably, due to it having a natural, intuitive meaning, it is rarely defined, which can lead to confusion [Par79]. For this thesis, each ‘layer’ corresponds to each level of the object composition hierarchy of a program written in an object-oriented programming language; or corresponds to the calling of methods – as they hide an arbitrary amount of complexity. For example, a Dictionary object is one layer of abstraction, while at the next lower level of abstraction there may be an implementing data structure such as a Hash Table or BTree object. The key aspect of a layer of abstraction is that it abstracts concerns present at a lower level so that the user at the higher level does not need to worry about them.

In object-oriented programming, inheritance can also be viewed as a system that allows levels of abstraction as parent classes are intended to provide a more abstract view of child classes. However, such abstraction is mainly helpful from the class implementor’s perspective – from the perspective of a user of a class it is seen as a single whole.

Software Dimensions represent a system of describing layers of abstraction at a finer level of granularity to distinguish between these two types of abstraction. The term ‘levels of abstraction’ is considered analogous to ‘layers of abstraction’.

![Figure 9.1: The three ‘Software Dimensions’](image)

Figure 9.1: The three ‘Software Dimensions’
9.1.1 The 1st dimension

A program without sub-procedures can be considered to be one-dimensional because all statements are executed within the same layer of abstraction. In order to be sure that the program is working correctly, the programmer must understand the effect caused by every line. Due to this, if lines are modified, there is increased potential for unexpected side effects.

It follows that a text editor can be described as being one-dimensional due to the current programming context only being able to comfortably show statements at one layer of abstraction at a time, i.e., the current method that is displayed. Although, a called method may be shown either above or below the current method, this is often not the case.

The relationship between the use of a variable – either local, member, or global – and its declaration does not fit neatly within the concept of Software Dimensions. As, from the programmer’s perspective, the relationship is often informational, for example, determining the type of a variable, and as such information can be easily represented using a text popup, such relationships are considered to belong to the 1st dimension.

9.1.2 The 2nd dimension – composition/methods

The use of methods and objects adds another Software Dimension as they enable a further level of abstraction – they allow an arbitrary number of statements to be considered as one statement, or an arbitrary number of objects to be considered as one object. For example, the following statement abstracts the mathematical calculations required to return the square-root of ‘30’:

\[ x = \text{squareRoot}(30); \]

A programmer does not need to understand how the \textit{squareRoot} method calculates the result, nor do they need to worry that changes to the current method might affect the result returned from the \textit{squareRoot} method. Similarly, in object-oriented programming languages, methods on objects provide an extra layer of abstraction and therefore are also considered to represent the second dimension. For example, the following statement provides a similar abstraction to the previous example:

\[
\text{Integer anInteger = new Integer(30);}
\]

\[
\text{x = anInteger.squareRoot();}
\]

As an object may be included as a class member and the method called in a similar way, composition is treated as belonging to the same dimension. In both cases, flow of execution passes from the current context to another.

\[
\text{this.integer = new Integer(30);}
\]

\[
\text{x = this.integer.squareRoot();}
\]
Similarly, if the members of an object are accessed, it is considered to represent the 2nd dimension:

```java
this.rectangle.width = 50;
this.rectangle.height = 100;
```

Access to a shared object, such as to a variable passed as a method argument or to a shared lock, is also considered to be represented by the 2nd dimension as it represents a layer of abstraction. Even though shared objects may exist in multiple places at once, to the programmer they only appear in the current location.

### 9.1.3 The 3rd dimension – inheritance

A third dimension is added through object polymorphism, which results from class inheritance and the use of interfaces. A single method call on a polymorphic object may cause multiple methods to be executed that are in the same layer of abstraction. This occurs if the method called is a constructor, a virtual method, or a method that calls a method of an ancestor class.

#### Constructors

When a statement instantiates an object, at one layer of abstraction an object is returned that is appropriately initialised, but at the next lower layer of abstraction, the object’s constructor may first call a parent constructor, to initialise the state of any data members defined within the parent class, before it initialises its own state. It is often necessary that the programmer fully understands what is happening in the calling context (where the object is instantiated) and in the called context (where multiple constructors might initialise the object).

#### Calls to ancestor methods

An overriding method is a method that has the same signature (has the same name and takes the same parameter types) as a method in an ancestor class. The reason that a method is overridden is because the functionality of the parent method needs to be altered to suit the child class. Often the overridden method is called and the returned value is used by the overriding method to prepare its own value. The calling of such a method may be considered analogous to the call of a parent constructor.

#### Virtual methods

A virtual method is a special method that is overridden, or implemented, by a descendant class. In Java, all methods are treated as virtual if overridden, while in other languages (such as C++), methods must be explicitly declared as ‘virtual’. When a virtual method is called, dynamic dispatch is used to determine the appropriate method to execute. Like constructors
and overriding methods, a virtual method might also call a corresponding method in an ancestor class. The key point is that if a virtual method is called, the call may be on an ancestor class or interface, while the methods executed are determined by the target object’s runtime type.

9.2 Supporting the ‘Software Dimensions’

It is often suggested that a programmer may simply manually arrange editors to see source code in a manner analogous to the Software Dimensions. This proves more difficult than one might expect due to modern IDEs, such as Eclipse and Xcode, constraining how files are presented within the environment’s window [B+10]. Xcode only allows one file to be open at a time in an Xcode window’s primary editor, however, recent versions of Xcode do provide companion editors, which can be displayed beside the primary editor and can automatically change based on the file in the primary editor – for example, it may be set to display the header file of a C-style language source file. Xcode now also provides tabbing support, but unlike most other development environments, each tab also includes their own auxiliary views, such as the left-hand file browser (Navigator). Although, double clicking a file in Xcode opens it in a separate window that may be manually arranged on the desktop, this causes contention for desktop space between these windows and the original Xcode window. In Eclipse, each file is opened in an editor tab. While a tab may be dragged to the side of the editor area to split the screen between two files, screen space soon becomes a limiting factor as the editor area is constrained. Like Xcode, you can drag tabs off the main Eclipse window so that they are displayed as independent windows, however, an additional constraint in Eclipse is that each source code file can only be opened in a single editor at once. This can cause issues if trying to arrange all methods that are part of a specific branch of the call graph if more than one of those methods is located in a single source file (which is often the case).

Describing the emergent structure of software in terms of dimensions guides how such dimensions might be represented on a 2D screen, or in an environment supporting method-flow. The second-dimension, which reflects method calls, can be represented by positioning a column view containing a method editor to the right and adjacent to the calling method. The third-dimension, which reflects polymorphism, can be represented by positioning a method either above (in the case of a parent constructor) or below (in the case of a virtual method) the method signature that is called.

9.2.1 Supporting the 1st dimension

It is thought that, currently, the first Software Dimension is adequately supported by current code editors that provide syntax highlighting and method completion, however, companion visualisations, such as control structure diagrams would also provide additional support for this dimension (see Section 5.2.5).
9.2.2 Supporting the 2nd dimension

The second Software Dimension may be supported by ensuring that method calls are represented as hyperlinks within the editor, and that following such a hyperlink causes a new editor to appear within a flow view. Figure 9.2 shows the same code as the example in Chapter 6 represented in a method-flow flow view. The calls to the constructors Rectangle and Circle, as well as the call to the areaOf method are represented as hyperlinks. The constructor of the Rectangle class is represented in an adjacent column editor within the flow view as if the Rectangle hyperlink had been selected.

As the flow view is a scrollable window, the call to super in the Rectangle constructor could also be clicked to cause the constructor for the Shape class to appear in another adjacent editor column. However, if the constructor were placed to the right of the Rectangle constructor this would not make logical sense because the Shape constructor executes before the Rectangle constructor. According to the Software Dimensions, the call to the constructor of Shape is really a call in the 3rd dimension. Therefore, calls in the third dimension need to be represented differently.

<table>
<thead>
<tr>
<th>Flow-View Window</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main.java</strong></td>
</tr>
</tbody>
</table>
| public static void main(String[] args) {
| \hspace{1cm} \hspace{1cm} Shape shape1 = new Rectangle(20, 40, red);
| \hspace{1cm} \hspace{1cm} Shape shape2 = new Circle(100, red);
| \hspace{1cm} \hspace{1cm} double area = shape1.areaOf() + shape2.areaOf();
| \hspace{1cm} \hspace{1cm} System.out.println("Area: "+ area);
| } |
| **Rectangle.java** |
| public Rectangle(int w, int h, String col) {
| \hspace{1cm} super(col);
| \hspace{1cm} this.width = w;
| \hspace{1cm} this.height = h;
| } |

Figure 9.2: Supporting the 2nd Software Dimension

9.2.3 Supporting the 3rd dimension

The third programming dimension is brought about through calls to parent constructors, overridden methods in an ancestor class, and virtual methods. An interesting aspect of object-oriented programming languages that support inheritance is that when a method is called, code in ancestor classes may also be implicitly executed before the called method. This typically happens during object instantiation so that the state of ancestor classes can be initialised before allowing an inheriting class to access their state. Similarly, sometimes overridden methods may call the method they override before doing anything else. A way of visualising such a scenario
is to display called ancestor constructors or methods above the called method. This might be achieved by splitting an editor column into two halves (one above the other). The benefit of this representation is that the execution of statements proceeds from the top of the screen down to the bottom of the screen. In the case of overloaded or virtual methods where such an ordering cannot be achieved because the method call is made in the middle of the method, an additional column editor might be added.

<table>
<thead>
<tr>
<th>Flow-View Window</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main.java</strong></td>
</tr>
<tr>
<td>public static void main(String[] args)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>Shape shape1</td>
</tr>
<tr>
<td>= new Rectangle(20, 40, red);</td>
</tr>
<tr>
<td>Shape shape2</td>
</tr>
<tr>
<td>= new Circle(100, red);</td>
</tr>
<tr>
<td>double area</td>
</tr>
<tr>
<td>= shape1.areaOf() + shape2.areaOf();</td>
</tr>
<tr>
<td>System.out.println(&quot;Area: &quot; + area);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

**Source code now visible in three dimensions**

Figure 9.3: Supporting the 3rd Software Dimension

### 9.3 Key design decision – method representation

Similar to when implementing method-flow (see Section 8.4.3), an important design decision to be made when implementing the Software Dimensions representation is whether to develop an implementation that uses whole source file code editors, or one that provides separate method editors. An advantage of showing the entire source file is that a programmer can browse the file to see other methods and comments. A disadvantage is that if the programmer scrolls away from the method involved in the navigated call graph, they may become confused and disoriented if they later return to the editor column and do not recognise it.

Providing support for the 3rd dimension is also problematic. As mentioned in the previous section, a column of the flow view could show multiple column editors – one above another. However, similarly, if these column editors showed more that one method editor, the programmer may become disoriented if either of these were scrolled to a different method.
9.4 Limitations

A limitation of the Software Dimensions is that it does not suggest a way of representing called-from links (back-links). Conceptually, this reveals an interesting problem – the composition hierarchy of software is actually only fully realised at runtime and consists of instantiated objects that each have individual state. However, in contrast, while navigating the structure of software, the programmer sees only the one individual source code file that represents all instances. If the programmer desires to modify a class, they may have one of two different intentions: the first is that all object instances will reflect the change at execution time; the second is that only one instance should reflect the change, which would require the creation of a new subclass. Resolving these issues is left for implementations.

9.5 A hypothetical cognitive map

The aim of the Software Dimensions concept is to confer a consistent spatial structure to the emergent structure of software so that during software navigations individual journeys can be integrated into a consistent cognitive map. It is important to remember that the visual representation provided by a computer display does not, and can not, dictate the representation of the programmer’s mental cognitive map. At best the computer representation acts as a muse by providing information in such a way to not confuse the existing representation.

For the point of discussion, Figure 9.4 proposes a possible form that a cognitive map may have after several navigations through the call graph of a piece of software using a method-

![Figure 9.4: A hypothetical cognitive map](image-url)
flow that supports the Software Dimensions. The arrow lines pointing right represent the call graph, the bold lines represent the composition hierarchy, and the upwards arrow between the Superclass and Object (2) classes represents the inheritance hierarchy. The cognitive map assumes that the programmer began navigating at the main method, and followed the call graph to the method of Object (1), and then to the constructor of Object (2). The programmer then scrolled the flow view back to the main method and followed the method call to Object (3), then again to Object (4).

Because all call graph branches originate in the main method it is mentally represented as overarching all other objects and methods. Similarly, method calls made from any particular method are overarched by that method. If an object is passed as an argument to a method, it would also be overarched by the method it was passed into. Due to the implementation supporting the 3rd Software Dimension, it placed the Superclass constructor above the constructor of Object (2) when that method was navigated to, resulting in the superclass constructor also being conceptually above the class’s constructor in the cognitive map. This representation has some similarities with the representation of the in-memory object composition hierarchy of PCVis [KS13].

If navigating software using a method flow that supports the Software Dimensions does form a cognitive map similar to that of Figure 9.4, the cognitive map representation should remain consistent as the programmer navigates more call graph branches. While navigating through another call graph branch may insert a new branch between two existing branches, existing branches should remain unaffected.

9.6 Summary

The connections between software declarations, which form the emergent structure of software, have semantic differences such as composition and inheritance. These semantic differences allow different layers of abstraction to be differentiated into different Software Dimensions that inform how software may be more intuitively displayed on a constrained two-dimensional screen. The first Software Dimension relates to the current method context. The second Software Dimension relates to the composition hierarchy, and call graph. The third Software Dimension relates to polymorphism. The second Software Dimension may be supported by adding a new editor column to a flow view, while the third Software Dimension requires manipulation of a column editor. A key design decision is whether to display an entire source code file, or whether to only display a single method, in each editor column. It is theorised that if an implementation of method-flow also supports the Software Dimensions, as a programmer navigates the call graph of a piece of software, their journeys will be integrated into a consistent cognitive map representing the software’s emergent structure.
Part III

Implementations

Two software tools, for the Java programming language, have been developed that implement the method-flow visualisation technique by supporting the 1st and 2nd Software Dimensions: the first, Code-Flow, is a tool for software exploration developed in Java; while the second, Visuocode, is a tool for program composition developed in Objective-C++. Rather than show methods within a traditional source code editor that shows the entire source file, these two implementations extract methods into separate method editors, which are then placed within the flow view’s editor columns. The rationale for this was to better support spatial memory by ensuring that methods are not susceptible to being scrolled vertically off-screen and therefore were more consistent within the flow view. As these implementations are not intended to be production ready environments they only support a subset of Java programming language; for example, they do not support inner classes, nor do they support class-style enumerations.
Chapter 10

Code-flow

An initial implementation of method-flow has been developed called Code-flow, which can be used for software exploration of Java source code.

Code-flow is available from: http://www.code-flow.com

10.1 Interface design

The interface of Code-flow was designed to be similar to the class browser interface of Visualworks Smalltalk and Visual Age for Java. Like these environments, the Code-flow window is split into two regions – the left-hand region is used for direct navigation, while the right-hand region holds the flow view. The left-hand direct navigation region is itself split horizontally into two regions: the top allows fully qualified classes to be selected, while the bottom allows specific methods to be selected, which are then displayed within the flow view. Each editor column in the flow view contains a header section that includes a label with the fully qualified class name, then a section that lists all of the class members. Any methods are displayed below the header section with method calls represented as hyperlinks.

Figure 10.1 shows a Code-flow window that has parsed a software project of mine called DanBox (a toy CPU emulator). The classes list shows a list of 10 classes, one of which is selected. The methods list shows the methods defined within the selected class. Selecting the class DanBox caused the first column to be displayed in the content area – this column shows the content of the DanBox class. Following the hyperlink-enabled ‘this.systemBoard.initBIOS’ method call caused the initBIOS method of the SystemBoard class to be displayed in an adjacent text editor column. Finally, the hyperlink-enabled ‘this.bios.init’ method call was followed, which caused the init method to also appear. A header at the top of each column shows the corresponding class name of each method, as well as the names and types of class members.
Figure 10.1: A Code-flow window with three source columns in a flow view.
10.2 Design decisions

Code-flow has a structured interface – direct navigations cause the flow view to be emptied and the selected class to be displayed in the flow view. Code-flow view columns do not have their own scrollbars to ensure that each editor column is visually distinct – meeting the first requirement for a visuo-spatial programming interface. Within the code area of each column, only method calls are marked up as navigation links. For example, following the ‘this.systemBoard.initBIOS’ link would add a view column for the SystemBoard class with just the initBIOS method. As methods are only added in an order that corresponds to adjacent methods on the call graph, the spatial consistency of methods is maintained – meeting the third requirement for a visuo-spatial programming interface. The spatial consistency is classed as permanent because as long as the codebase represented does not change, any call graph branches represented in the flow view remain spatially consistent. Unfortunately, Code-flow does not provide any explicit landmarks to aid navigation – potentially navigation through specific packages could be highlighted by enclosing classes within a package representation, and using colour to identify different packages.

10.3 Architecture

Code-flow is implemented in the Java programming language. The implementation contains three main modules: CodeFlow provides the user interface; AST provides source code parsing functionality; and OpenXDS provides operating system support, as well as common algorithms and data structures. The Code-flow user interface is implemented using Java Swing interface components including: JSplitPane, JList, JPanel, and JComponent. As Code-flow is purely a software exploration tool, and no editor was required, method source code is implemented by drawing on a JComponent view. The AST module implements a source tokeniser and parser that provides an abstract syntax tree to the Code-flow SourceModel class. OpenXDS is an open source cross-platform development suite of libraries that the author has implemented in Java and C++. In addition to providing a common method of interacting with the operating system, OpenXDS also provides algorithms and data-structures that are not provided by the Java class library.

10.4 Limitations

As Code-flow was implemented as a proof-of-concept for the method-flow visualisation technique it has some limitations. Within Code-flow’s flow view, each source column only presents the name of the class, the class members, and any methods that should be shown – other source code declarations such as imports and enumerations are not shown. Due to Code-flow being a
read-only tool, it was decided to parse all source files into one large abstract syntax tree during initial parsing to simplify the implementation, however, this makes it difficult for the system to be modified to respond to source code changes made by other programs. Another limitation is that only one top-level source directory can be added to the system.

When an initial direct navigation is performed, the user may select either a class or a method. If a large class is selected, it makes the flow view correspondingly tall, which means that when a method is then followed, that method is represented at the top of the flow window making it difficult to view the initial method and the called method at the same time. Similarly, if the class of one view column contains many more class members than the class of an adjacent view column, the list of class members causes the two methods to be offset vertically.

10.5 Summary

Code-flow is a prototype Java software exploration tool that implements the method-flow visualisation technique that was implemented in the Java programming language. Its interface is styled after the class browser interface of Smalltalk and IBM Visual Age. As the implementation parsed Java into one abstract syntax tree, which made it impractical to respond to changes to source code from other programs, or implement editing functionality, the implementation would have required major alteration if it was to be extended.
Chapter 11

Visuocode

Visuocode is a second implementation of the method-flow visualisation technique that supports the first and second Software Dimensions. Like Code-flow, Visuocode supports the Java programming language. In contrast to Code-flow, Visuocode has been developed specifically to support software composition.

Visuocode is available from: http://www.visuocode.com

11.1 Interface design

Visuocode uses a multi-window paradigm that consists of two types of window: the Workspace Manager window, which is used to manage software projects; and associated Flow windows, which contain method-flow flow views. A multi-window paradigm allows multiple flow view windows to be arranged across multiple displays. The use of the terminology ‘workspace’ and ‘project’ were chosen to be analogous with other IDEs.

11.1.1 The Workspace Manager window

The Workspace Manager (see Figure 11.1) allows the programmer to manage a collection of projects. Each project represents a collection of one or more Java packages that share a common class path. A simple workspace might contain only one project while others may link to many. Projects may be shared by any number of workspaces.

When a project is added to a workspace its Java source files are parsed and a subtree containing its packages, classes, and method signatures is added beneath the ‘Workspace Projects’ root node of the Workspace Manager window. Selecting either a class or method causes a Flow window to appear with a new editor column containing either the selected method or, if a class was selected, all the methods of that class. If a Flow window is already present, its content is replaced by a new editor column corresponding to the selected method or class. Optionally, Visuocode may be configured to open methods and classes selected in the Workspace Manager in a new Flow window.
11.1.2 The Flow window

A Flow window contains a flow view that contains one or more editor columns. Each editor column is associated with a specific class. Initially, the Flow window contains a single column corresponding to a method or class that was selected from the Workspace Manager.

Each editor column contains five separate areas. The first four, collectively referred to as the class attributes area, are: an editable class/interface declaration field; an editable class imports list; an editable enumerations list; and an editable members list. The last area, referred to as the methods area, contains a scrollable stack of one or more method editors. If the column editor was opened by clicking a class name in the Workspace Manager, there is a method editor for each method of the class.

Within method editors, resolving method calls are represented as blue hyperlinks – if followed, an editor column for that method is inserted to the right of the existing column. Resolving classes are represented as crimson hyperlinks – if followed, an editor column for that class that contains all the methods of that class is inserted to the right of the existing method-flow. As all columns sit within a scrollable view, the programmer can scroll back to the left to view obscured column editors.

Within a flow window, the widths of the column editors may be adjusted by dragging the column separators that separate them. This allows intervening editor columns to be collapsed to juxtapose editor columns that are non-adjacent on the call graph – such as if delegation is being used.

Figure 11.1: Visuocode consists of two types of window – the Workspace Manager window (left column), and the Flow window (remaining columns).

The screen-shot was taken from a small (laptop) screen for typesetting purposes.
Figure 11.2: Visuocode Flow window – the programmer has navigated from the main method, through the WinMain, and ProcessFiles methods, to the ProcessCSVFile method; within ProcessCSVFile they have selected the csv file local object causing an editor column for the CSVFile class to be displayed that shows all of its methods.

The screen-shot was taken from a 27” iMac with Retina 5K display.
11.1.3 Support for program composition

Visuocode was specifically implemented to provide extra support for program composition. Method calls that fail to resolve are represented as red hyperlinks. If followed, the method call is analysed to determine which class the new method should be inserted into, then an editor column containing a method skeleton is displayed. For example, a non-resolving implicit method call, or a method call explicitly invoked on ‘this’, would be inserted into the current class, e.g.,

```java
int x = getValue();
```

or

```java
int x = this.getValue();
```

If a non-existent method is invoked upon a specific object, it would be inserted into the class corresponding to that object. If a newly created method is saved, it is inserted into the source code file at the bottom of the class. For example, assuming that a `getArea` method has not yet been defined in the `Square` class that takes no parameters; following the `getArea` method call would display a column editor for the `Square` class that contains an appropriate method skeleton:

```java
Square aSquare = new Square( 10 );
int x = aSquare.getArea();
```

If a non-existent method call includes valid arguments, the types of those arguments are used to determine the signature of the method skeleton, however, the current implementation is currently unable to guess what the return type should be based on the surrounding context.

In situations where dynamic dispatch is being used, the method to be created may be intended for another class within the class hierarchy of the target object. To create such a method, the intended class needs to be explicitly specified. For example, assuming that the `Square` class extends the `Shape` class, a method can be created in the `Shape` class by explicitly calling `getArea` on `Shape`:

```java
int x = Shape.getArea();
```

Once implemented and saved, the line can be rewritten as:
Non-resolving class types are hyperlink-enabled and coloured in grey. Following such a link will cause an “Add New Source File...” dialog to appear that allows the user to indicate a project, package, classname, and initial method signature, then create the corresponding source file. Once a class’s source file is created, methods may then be added to it using the process described above, or by writing a new method beneath an existing method within its code editor.

11.2 Design decisions

Visuocode has a structured interface – direct navigations cause the flow view to be emptied and the selected class to be displayed in the flow view within an editor column. Unlike Code-flow, within each editor column, each list, and the methods area now has a vertical scrollbar. Each editor column now contains a horizontal scrollbar, and the width of individual columns may be resized. This is particularly useful if a class is opened that has one or more extremely wide methods. Additionally, the class label, imports list, enums list, and members list now all have default heights that are consistent between all editor columns, which ensures that, unless their heights are altered by the user, the methods area for each column lines up horizontally. As navigated methods remain visually distinct, the first requirement for a visuo-spatial interface is meet. With each code editor, class types, members, and variables are now all hyperlink-enabled, allowing navigation of both the call graph and the composition hierarchy. As classes or methods are only added in an order that corresponds to path a through the emergent structure of a piece of software, the spatial consistency of methods is maintained – meeting the third requirement for a visuo-spatial programming interface. The spatial consistency is classed as permanent because as long as the codebase represented does not change, any call graph branches represented in the flow view remain spatially consistent.

As Visuocode was specifically developed to support program composition, design decisions were made that related to developing software. Due to Visuocode not presenting whole source code files, the system needed to ‘decide’ where to put any newly created methods. As the Workspace window shows the methods of each class alphabetically (uppercase letters before lowercase), it was decided that there was no need to intelligently place newly created methods within the destination source code file, and that adding them to the bottom was sufficient. Similarly, imports, enums, and members, are added by editing the ever-present empty last item of each list. Such items are saved directly below the last item of the same type; for example, a class member is saved directly below the last class member.

An important decision that was made was to make links navigable by default so that relative navigation would be obvious. This decision is particularly important for a development environment because programmers often want to click on a class, variable, or method call in
order to edit its text, not to navigate elsewhere. In order to edit such a link it is necessary to position the cursor to the left or right of the link.

A key goal of the Visuocode environment was to make the creation of new software structure as easy as possible in order to encourage higher quality software structure. To this end, the creation of new methods was explicitly supported by being able to follow non-resolving methods, which causes a new method skeleton to be created within the target context. In hindsight, one problem with this approach has been identified – if the programmer wishes to call an overloaded method, if the arguments that they are passing do not match an existing method, they may become confused when an empty method skeleton is shown.

Although not emphasised, a programmer may also navigate to the desired class, and add a new method below an existing method. Potentially, another way of adding a new method is to have a non-resolving link popup a dialog, very similar to the ‘Add Source File’ dialog, that creates a new method, and then inserts it in the desired class, however, this was felt to break the relative navigation paradigm that Visuocode is emphasising – in particular, the question arises as to what the environment should do if what the programmer has already written conflicts with the information entered into the dialog. One advantage to the current system is that the method is not saved into the target file immediately, however, if the programmer is unaware they need to save such methods there is a potential for loss of information.

11.3 Limitations

As Visuocode uses static analysis to resolve method calls it is often not possible to determine the runtime type of various objects. This can cause particular trouble if using frameworks that make extensive use of interfaces. A possible solution is to determine what classes extend or implement such interfaces and provide a contextual menu that allows the user to select the desired class.

Another limitation of Visuocode is that it does not provide a mechanism for identifying called-from links (methods that call the current method). It was decided not to implement such functionality because, firstly, it would have caused too much disruption to the implementation of the Visuocode user interface, and, secondly, because it was felt that it was preferable to empirically evaluate Visuocode before such functionality was added.

11.4 Unresolved issues

In order to create the coloured hyperlinks, method source code must be parsed so that any method calls can be resolved, or found non-resolvable. During the implementation of Visuocode, it was decided that method calls would be resolved when each editor was redrawn. The rationale for this was that editors would only need to be redrawn if code was modified, and that any method calls would need to be re-resolved because related variable declarations may have
changed. Once saved, any visible methods calling that method would also need to be resolved again as the method’s signature may have changed. Unfortunately, while this approach worked well for codebases with small classes, if classes contained numerous methods, and if all the methods of a class were displayed, method call resolution caused a significant lag. To mitigate the lag, caching of method call resolution was implemented, however this was unable to prevent the lag experienced the first time a class was viewed. An alternate strategy, that is being considered, is to pre-resolve all method calls before the Visuocode window is presented, so that lag is minimised when a class is opened for the first time.

11.5 Implementation rationale

11.5.1 Why not an Eclipse plug-in?

Due to the popularity of the Eclipse development environment, before the implementation of Code-flow, the feasibility of building it as an Eclipse plug-in was investigated. This was decided against due to experience reports [LMSW03] that suggested that adapting the Eclipse user interface would be difficult, and that there was a possibility that the existing text editor functionality could not be easily reused. In addition, an Eclipse plug-in would require continuing maintenance to keep it up-to-date with the evolving Eclipse platform.

11.5.2 Why write your own parser?

After deciding not to implement as an Eclipse plug-in, several parser generation systems were investigated such as ANTLR and Bison. It was decided not to use these as they did not generate the parser source code in the required programming language (at that time Java), stripped out whitespace, which is required for editor functionality, or required a Java language specification that was at that time unavailable. The Eclipse parser was also investigated, however at that time, it proved too difficult to extract from Eclipse. Therefore I implemented my own tokeniser and parser, called Astral.

11.5.3 Why port to Objective-C++/Cocoa?

After the initial proof-of-concept evaluation of Code-flow, the decision needed to be made whether to extend Code-flow or create a new application. Due to the limitations described in Section 10.4, and the desire to be able to distribute any new application via the newly announced Mac App Store, I decided to develop Visuocode in a combination of Objective-C++ and C++. The Code-flow AST module was able to be quickly ported to C++ as the original code used a suite of libraries that was implemented by the author in both Java and C++, called OpenXDS. Additionally, the user interface code needed to be rewritten in order to support the workspace/project paradigm, as well as support program composition. Unfortunately, after the
system was implemented Apple introduced new terms for releasing software via the Mac App Store that required that applications operate in a sandboxed environment that did not allow access to arbitrary files – Visuocode requires the ability to open any file that is located beneath a project directory. It is hoped that in the future Apple will provide a suitable mechanism to allow this functionality within their sandbox. However, Visuocode can still be easily downloaded, then run outside of the sandbox by right clicking on the Visuocode application and selecting ‘Open’.

11.6 Comparison to existing visuo-spatial environments

In Chapter 7, a visuo-spatial programming interface is defined as having to spatially separate visually distinct programming artefacts that are placed within a spatially consistent environment associated with a programming landmark for some period of time. Note that there is no prescribed manner that such an environment must go about supporting spatial memory.

The Visuocode environment is very different from the environments described in Chapter 7 as being visuo-spatial environments. Environments such as Code Thumbnails, Code Canvas, and Code Bubbles each provide a desktop metaphor that allows code fragments to be manually laid out with a spatial orientation. Primarily, the intent of these environments is to support the spatial navigation of the programmer amongst a set of files laid out on the desktop – though Code Canvas and Code Bubbles do allow related methods to be juxtaposed. As the spatial arrangement of software fragments should remain consistent, the environments allow the programmer to form a spatially consistent mental model of the software. Navigation is also supported by groups of fragments sharing a common coloured background to provide a landmark for the group.

In contrast, Visuocode provides a flow view window that contains editor columns. As this flow view is replaced if the programmer performs another direct navigation from the workspace window, one might think that the editor columns do not persist long enough for Visuocode to be classed as a visuo-spatial environment. However, a key aspect of Visuocode is that column editors are not manually arranged – column editors are added to a flow view based on relative navigation of the emergent structure of the software. Therefore, if a programmer repeats a particular traversal of the call graph, the column editors are laid out in exactly the same orientation as they were before. Therefore the orientation of editor columns within the flow view remains consistent across programming sessions, and only changes if the software structure changes.

The difference between the spatial navigation environments and Visuocode can be thought of as the difference between navigating a city using a map – or a birds eye view of a scene – and navigating by walking through the city. Spatial navigation environments provide a map that allows the programmer to quickly navigate to a specific location, while Visuocode allows the programmer to navigate by remembering – from spatial memory – where to go next in order to get where they want to go. Visuocode still requires the programmer to navigate within the
flow view, but also allows the programmer to quickly retrace their steps through the methods they have traversed by scrolling the flow view.

From the visuo-spatial programming perspective, method editors within editor columns provide a visually distinctive representation of programming artefacts, which are spatially separated within a flow view, which is the environment. The spatial arrangement of editor columns within a flow view is consistent for any specific path through the emergent structure of the software, and are therefore permanent between sessions. A potential criticism, however, is that Visuocode does not provide sufficient support for landmarks.

11.7 Informal evaluation

The goal of informal evaluation is to identify issues with software that might affect later formal evaluations. Different types of issues may be discovered during such an evaluation, including usability issues, domain-specific issues, missing functionality, and programmatic errors (bugs) that only occur in rare circumstances – detected bugs were fixed and are not discussed here. This chapter presents issues that were encountered during informal self-evaluation of Visuocode, or were raised by expert programmers as feedback. Initial evaluation was carried out by the author over several sessions with no specific tasks or materials. Activities included navigating and modifying various programs written in the Java programming language. The following sections report and discuss each issue encountered then describe the corrective action taken, if any. Expert feedback was solicited from advisors, faculty staff, and other students.

11.7.1 Issue 1: not being in control

Issue description

While composing source code, a feeling similar to not being in control was experienced. It is thought this was due to not having the expected complete control over the source file being edited. For example, if using a whole source file editor, a new member or a stub for a new method may be quickly inserted into the file. In contrast, if using Visuocode, unless a new method skeleton is created, a new method may only be entered within an existing method editor below the text of the existing method. When the editor is saved, the additional method is extracted leaving only the existing method. This feeling lessened after a short period of time as more concentration was directed at the programming task, and less on the environment.

Discussion

While programming, programmers form a mental model of software. Because they are used to interacting with whole source code files, programmers may also be used to forming their mental models in terms of whole source files. Providing an alternate representation, where source code
files are presented as fragments, may initially interfere with their established cognitive processes for constructing software mental models, resulting in an uncomfortable feeling. Such a feeling may also be heightened by an initial lack of trust in the environment as the programmer is unable to scroll around a source code file to reassure themselves that the code they have written is still there. This feeling was not identified during informal evaluation of the Code-flow software exploration tool, which suggests that it may be more related to program composition.

Corrective action

As the aim of this project is to investigate the effects caused by the use of the method-flow paradigm, no corrective action is to be taken.

11.7.2 Issue 2: handling polymorphic objects

Issue description

While navigating the source code of an application that used an application framework, it was found that Visuocode would only navigate to interface definitions, not the classes that implemented them as might be expected by a user.

Discussion

Visuocode determines what method a call should navigate to through a static analysis of the codebase. Polymorphic objects, whose declared type is that of an interface or a parent class, hide what the actual runtime type of the object is. While it may be possible to track the types of some polymorphic objects that are instantiated deterministically (as opposed to those whose type is determined at run-time), it is currently considered out of scope for this project to attempt such an implementation. An alternative strategy is to determine the classes that implement or extend such classes, and provide the user with a context sensitive list of possible class choices, however, this was considered out of scope for the project because of the interface rework that would have been necessary. This issue was identified during informal evaluation of the Code-flow software exploration tool, but was considered to be out of scope as the severity of the issue seems to be related to how interfaces are used. If an interface represents an abstract data type (such as a dictionary) there is far less need to navigate to the actual implementation, and it is possible that navigating to the interface definition is both expected and useful. In contrast, where software uses interfaces to expose limited aspects of classes, lack of support for polymorphism can lead to confusion, as the programmer is unsure of exactly what object is being interacted with.

Corrective action

No corrective action to be taken.
11.7.3 Issue 3: locating compiler errors

Issue description

During a programming task, it was difficult to trace the source code line number provided by a compiler error message to the source code displayed within Visuocode because the methods displayed within the column editor were not in the same order as in the original source file, and were also unnumbered.

Discussion

Programmers using Visuocode may be negatively affected by the environment not supporting incremental compilation as it is a standard feature in most modern development environments. Specifically, incremental compilation allows the environment to highlight syntax errors within the code editor, as well as provide links to the location of the error. This issue was not identifiable during informal evaluation of the Code-flow software exploration tool due to it not supporting program composition.

Corrective action

As it is expected that study participants will compile code using an external command line tool during the second study, line numbers have been added to method editors, and methods are now arranged within an editor column in the same order as in the source code file.

11.7.4 Feedback 1: lack of global or local search

Description

Several people have commented that they make extensive use of functionality that allows them to search source code for keywords. Visuocode, does not currently support keyword search.

Discussion

For the planned evaluations, keyword search is only likely to be needed during the study that investigates software navigation, as during studies of software composition there will not be enough existing code to need search capability. As the first formal study is intended to investigate navigation, and not search, the lack of this capability is not considered an issue, however, as Eclipse does support this style of search, the usage of search is to be analysed and considered as a threat to validity.

Corrective action

No corrective action to be taken.
11.7.5 Feedback 2: reverse navigation up call graph

Description

It has been suggested that Visuocode should also support reverse navigation up the call graph, as well as navigation down the call graph.

Discussion

It is agreed that this would be a useful feature, however, due to the amount of user interface rework required, this will not be implemented for the upcoming formal studies.

Corrective action

No corrective action to be taken.

11.7.6 Feedback 3: viewing Javadoc documentation

Description

It has been suggested that programmers often want to view just the Javadoc of source code, not the code itself.

Discussion

Currently, if a followed method call resolves to a standard library class, Visuocode opens the Javadoc corresponding to that class; whereas, if the method call is on a class whose source code is available, Visuocode opens that source code file in an editor column. While it is acknowledged that this would be an important feature to have for production use, it is not a feature that is likely to be relevant during the formal studies.

Corrective action

No corrective action to be taken.

11.8 Summary

Visuocode is a prototype Java development environment for Mac OS X that was implemented in Objective-C++. In contrast to Code-flow, the interface is composed of two window types – the Workspace Manager window and the Flow window – so that multiple flow views can be arranged on the screen. Visuocode provides similar direct navigation capabilities to mainstream development environments, while providing additional support for relative navigation by implementing method-flow. Unlike existing programming environments that are recognised as
having visuo-spatial programming interfaces, Visuocode does not provide a desktop metaphor. Instead, Visuocode relies on editor columns being added to a flow view according to the emergent structure of the software to ensure that the spatial arrangement of editor columns remains consistent.
Part IV

Formal evaluation

It is now expected that researchers empirically support the claims they make regarding the software tools they develop. However, researchers have found quantitative evaluation of software development tools difficult due to the variability of programmer performance on different tasks confounding statistical analysis. Due to this, it was decided to evaluate Visuocode using a qualitative methodology. This part describes the qualitative evaluation of the Visuocode prototype development environment.

As the concept of visuo-spatial programming has been introduced within this thesis, the evaluations described in this part are intended to provide a baseline for examining the visuo-spatial programming support provided by a traditional development environment – Eclipse – and the prototype development environment developed during this project – Visuocode. It is thought that if a programmer uses a development environment that provides additional support for relative navigation, their navigation behaviour will change to include more relative navigations than otherwise, which is considered to be a form of second degree evidence for the increased use of a spatial cognitive map.

In empirical studies of programmers, programming tasks can be categorised as being of software understanding, software maintenance, or program composition. Software understanding tasks usually involve navigating and comprehending an unfamiliar codebase, and then later being measured on the ability to answer questions concerning the codebase. Software maintenance tasks are similar to software understanding tasks except that the participant is also asked to make some corrective (fixing a bug) or perfective (adding a feature) changes to the codebase, and are also measured on the success of those changes. In contrast, program composition tasks usually involve creating a software program from scratch unconstrained by existing code. In the past, attempting to increase real world relevance, many empirical studies of programming tools have been carried out that involve either software understanding or maintenance tasks because these allow the use of a pre-existing, real world codebase. However, due to this, there are few studies that investigate the behaviour of programmers creating new software from scratch, and little is known about how the functionality provided by development environments affect the creation of software structure.
In order to better understand the difference in programmer behaviour while they attempt tasks on an existing codebase, as well as tasks that involve new code creation, this part describes two studies. Chapter 12 describes the experiment plan used for both studies. Chapter 13 describes a study that investigates programmer behaviour motivated by software understanding tasks. Chapter 14 describes a study that investigates programmer behaviour motivated by program composition tasks. These studies received ethical clearance from the University of Queensland Ethics Committee (ethical clearance reference EC201208BRA).
Chapter 12

A qualitative experiment plan

Once a controlled experiment has been chosen as the method of evaluation for a tool, it is necessary to develop an experiment plan that describes: the treatments (the combinations of condition and task) that are to be compared, the type of participants required, the type of data to be collected (quantitative or qualitative), the dependent variables to be analysed, the experimental design, and the analysis method. The experiment plan, in turn, influences the number of participants that are required for the study. Quantitative experiments require more participants in order to increase the likelihood of finding statistically significant results, while qualitative experiments that do not rely on a statistical analysis are able to be carried out with fewer participants.

The studies described in the following two chapters investigate the behaviour of participants while attempting each of two programming tasks using each of two software development environments. Qualitative data was collected in the form of screen-capture recordings. Both studies use a within-subjects, counter-balanced experimental design. Due to using a qualitative methodology, only a small number of participants were required – six participants took part in the first study, and five participants in the second. The screen recording from each participant session was analysed using protocol analysis to ‘code’ individual software navigations. The difference in the number of navigations of each kind performed by each participant was statistically analysed using repeated measures t-tests to determine if the difference in navigations while using each environment was statistically significant.

This chapter describes the rationale for the experiment plan. Section 12.1 discusses the treatments. Section 12.2 discusses the type of participant required. Section 12.3 discusses the data collected. Section 12.4 discusses the dependent variables that were quantified from the data. Section 12.5 explains the experimental design. Section 12.6 describes the statistical analysis. The main contribution of this chapter is the protocol analysis [ES84] methodology used to ‘code’ programmer navigation behaviour into individual navigations for analysis.
12.1 Treatments

The treatments to be investigated are each of the two development environments – Eclipse and Visuocode – combined with each of two, study dependent, tasks. While this plan compares the use of an experimental condition (a prototype development environment) against a control condition (a mainstream development environment), it may also be used to perform an exploratory study of a single condition. Several exploratory studies have been reported that have described programmer navigation behaviour while using the Eclipse IDE [RCM04, KAM05, dAM06].

12.1.1 Task choice

Two studies were carried out to evaluate the Visuocode software development environment. Study 1 was an investigation of software navigation, and Study 2 was an investigation of program composition. Each of the two tasks for Study 1 involve navigating the source code of a different piece of open source software obtained from SourceForge, while each of the two tasks for Study 2 involve composing new source code that implements a Java abstract class specification. Task choice is a critical aspect of evaluating software tools because programming is a cognitively challenging activity, and real world applications can contain hundreds, thousands, or millions of lines of code. Due to real world sized programs being infeasible for controlled studies, evaluation tasks described in the literature have often either used modification tasks on an existing unfamiliar codebase or small program composition tasks. However, such tasks have problems with generalisability. Modification tasks have the problem that the participant is unlikely to feel any sense of ownership of the code in question, and is unlikely to perform any restructuring of the code – in short, they may hack the code to complete whatever task they have been set. On the other hand, small program composition tasks are not considered ‘big’ enough to lead to generalisable results as participants are likely (perhaps without realising it) to develop the software in a manner different to how they would normally, such as not extracting functionality into separate methods or classes when appropriate. It is hoped that requiring participants to complete the implementation of a class that extends an abstract class is more similar to a real world work task than creating an equivalently sized, but complete, program, and therefore provides more generalisable results.

12.2 Participants

Due to Visuocode being intended for experienced programmers, only experienced programmers were accepted as participants. Early empirical studies of programming continued the psychological tradition of using undergraduate students as participants for experiments, unfortunately, this often led to confounded results because it was found that such participants had a widely varying level of performance [STW67, She81]. It is now accepted that novice programmers
should only be used in studies that evaluate tools intended for novices. For this thesis, experienced programmers are defined as people who identify as programmers, are not undergoing formal programming training, and are either employed as a programmer or are actively programming as part of their ongoing education (such as a graduate degree). In other words, they have either been taught or have taught themselves how to program, and are currently an active programmer.

12.3 Data collected

Qualitative data is collected by recording each participant session using screen-capture software that also records audio. A benefit of collecting qualitative data is that the researcher is better able to understand how each participant uses each environment, and is able to identify any issues that might pose a threat to the validity of the results. During the first study, over-the-shoulder video was also recorded but, as it provided little additional information beyond the screen recordings, it was not collected during the second study.

12.4 Variables

The dependent variables analysed are the number and type of navigations performed, which are obtained by performing protocol analysis [ES84] to code each programmer’s navigations during each task. Screen recordings are first analysed to determine the different types of navigations that are performed by the participants. Next the screen recordings are coded into a session transcript that records each navigation with a timecode, the navigation type, and free-form text that usually indicates the method navigated to – similar to the technique described by Robillard et al. [RCM04]. Table 12.1 provides a summary of the codes used for navigations. The codes are grouped by the four kinds of navigation: implicit, direct, relative, and scrolling, as well by navigations related to editing code.

12.4.1 Implicit navigations

Implicit navigations are characterised by a programmer moving their focus from one visible method to another visible method. This usually occurs when the programmer moves to another method within the same source file, which is referred to as a LOCAL navigation (→), or when the programmer swaps from one visible editor to another (or from one visible window to another), which is referred to as a lateral navigation – FOCUS LEFT OR RIGHT (← or →). Often, an editor becomes visible if another editor tab is closed – this is referred to as a CLOSE navigation (🗑). For want of a better term, an EQUIV navigation (≡) occurs if a method becomes visible because it is in the same vicinity of a source code file to the method the
programmer just navigated to. EQUIV navigations are not included in the analysis, but are included in the session transcripts to provided a richer account of what occurred.

12.4.2 Direct navigations

Direct navigations are characterised by a programmer navigating directly to a location in a file using an auxiliary view. Often, these are performed using a file browser, such as the Eclipse Package Explorer or the Visuocode Workspace Manager, and are referred to as FILE

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>LOCAL</td>
<td>The participant noticeably changes focus to another already visible method.</td>
</tr>
<tr>
<td>←</td>
<td>FOCUS LEFT</td>
<td>The participant switches focus leftwards to a window or tab-pane.</td>
</tr>
<tr>
<td>→</td>
<td>FOCUS RIGHT</td>
<td>The participant switches focus rightwards to a window or tab-pane.</td>
</tr>
<tr>
<td>⊙</td>
<td>CLOSE</td>
<td>An editor/method is made visible when tabs are closed (Eclipse only).</td>
</tr>
<tr>
<td>≡</td>
<td>EQUIV</td>
<td>The corresponding method is also on screen after a navigation.</td>
</tr>
</tbody>
</table>

FILE A file navigation using the Project Explorer or the Workspace Manager.

AUX A navigation using an auxiliary view, such as the outline view.

SEARCH A navigation after searching for text within a source file.

BACK A backward navigation using the Eclipse back button (Eclipse only).

FORWARD A forward navigation using the Eclipse forward button (Eclipse only).

TAB LEFT Navigating to an editor tab to the left.

TAB RIGHT Navigating to an editor tab to the right.

CALL UP A navigation up the call hierarchy using the Eclipse Call Hierarchy view.

CALL DOWN A navigation down the call hierarchy using the Eclipse Call Hierarchy view.

REL FORWARD A forward navigation down the call hierarchy.

REL LEFT A navigation back up a displayed call graph branch (Visuocode only).

REL RIGHT A navigation back down a displayed call graph branch (Visuocode only).

SCROLL TOP Scrolled to top of file.

SCROLL BOTTOM Scrolled to bottom of file.

SCROLL UP A navigation to a method by scrolling up.

SCROLL DOWN A navigation to a method by scrolling down.

IMPORT The participant creates or edits a class import.

MEMBER The participant creates or edits a class member.

EDIT The participant creates or begins editing a method declaration.

Table 12.1: Symbols representing navigation behaviour codes
navigations (⊕). Alternatively, direct navigations may result from performing a text SEARCH (○), or using another view such as the outline view (⊙). The ◀ and ▶ symbols are used to indicate when the participant has used the history buttons to navigate BACK or FORWARD in their history. While using environments that support tabbed editors, the programmer may swap between a number of open tabs – these are referred to as either TAB LEFT or RIGHT navigations (← and →).

12.4.3 Relative navigations

In contrast to direct navigations, relative navigations are characterised by a programmer navigating the emergent structure of software, such as its call graph. As relative navigations are performed differently in different development environments, they are also coded differently to distinguish between the functionality used. Eclipse users can use the Eclipse Call Hierarchy view to discover the methods that call a selected method, as well as the methods that it calls – navigations made using this view are referred to as CALL UP and DOWN navigations (⇐ or ⇒).

Method calls in Visuocode are automatically hyperlink-enabled, while method calls in Eclipse may be hyperlink-enabled by holding down a modifier key and hovering the mouse over the method call. Following a hyperlink-enabled method causes that method to appear – in the case of Visuocode, in an adjacent editor column; whereas in the case of Eclipse, a new tab pane containing the editor is opened. In both cases, such a navigation is referred to as a RELATIVE FORWARD navigation (⇒). To return to the previous programming context, if using Visuocode, the programmer can scroll back and forth within a flow view by performing RELATIVE LEFT and RIGHT navigations (← and →), however, if using Eclipse, the programmer must either use the Call Hierarchy view or the history back button.

12.4.4 Scrolling navigations

Scrolling navigations are characterised by a programmer scrolling vertically between methods in a file editor or within the methods area of a Visuocode editor column. Programmers exhibit two kinds of scrolling behaviour: slow scrolling, such as when scrolling down through each and every method in a source file; and fast scrolling, such as when a programmer scrolls quickly to the top of a file. When a programmer scrolls between individual methods these are referred to as SCROLL UP and DOWN navigations (↑ and ↓), and when coded include all recognisable methods that are scrolled over, whereas when a programmer quickly scrolls to the top or bottom of a file these are referred to as SCROLL TOP and BOTTOM navigations (↑↑ and ↓↓).
12.4.5 Editing behaviour

An IMPORT edit ($i$) occurs when the programmer modifies an import statement. A MEMBER edit ($m$) occurs when the programmer modifies a class member. An EDIT ($\Delta$) occurs when the programmer creates or modifies a method.

12.4.6 An example session transcript

Table 12.2 shows a partial section of an example session transcript of software navigation that has been converted into a table. The source data is collected into a comma separated value (CSV) file whose first field is a timecode, second field is a Latex symbol identifier, and third field is free-form text that describes the navigation. A command-line software tool has been developed that automatically processes session transcripts to produce a contingency table also in CSV format. Contingency tables are used in statistics to present categorical data. For example, Table 12.3 shows the number and type of navigations extracted from each session transcript of a single participant. Each of the rows that start with "P1" corresponds to a session transcript. Table 12.4 is a contingency table that summarises the navigations based on the kind of navigation performed.

<table>
<thead>
<tr>
<th>Time</th>
<th>Type</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:05</td>
<td>$\oplus$</td>
<td>Main.main()</td>
</tr>
<tr>
<td>00:10</td>
<td>$\oplus$</td>
<td>ClientManager</td>
</tr>
<tr>
<td>00:10</td>
<td>$\equiv$</td>
<td>ClientManager.ClientManager()</td>
</tr>
<tr>
<td>00:30</td>
<td>$\sim$</td>
<td>ClientManager.connect()</td>
</tr>
<tr>
<td>00:35</td>
<td>$\Rightarrow$</td>
<td>ClientManager.connect() $\Rightarrow$ ClientScreen.replyReceived()</td>
</tr>
<tr>
<td>00:50</td>
<td>$\Rightarrow$</td>
<td>ClientManager.connect() $\Rightarrow$ ClientScreen.replyReceived() $\Rightarrow$ doLogin()</td>
</tr>
<tr>
<td>00:50</td>
<td>$\equiv$</td>
<td>ClientManager.connect() $\equiv$ ClientScreen.replyReceived() $\equiv$ doLogout()</td>
</tr>
<tr>
<td>00:55</td>
<td>$\sim$</td>
<td>ClientManager.connect() $\sim$ ClientScreen.replyReceived() $\sim$ doLogout()</td>
</tr>
<tr>
<td>00:59</td>
<td>$\sim$</td>
<td>ClientManager.connect() $\sim$ ClientScreen.replyReceived() $\sim$ doLogin()</td>
</tr>
<tr>
<td>01:05</td>
<td>$\Leftarrow$</td>
<td>(opens Call Hierarchy for ClientScreen.doLogin())</td>
</tr>
<tr>
<td>01:15</td>
<td>$\Leftarrow$</td>
<td>ClientScreen.actionPerformed $\Leftarrow$ ClientScreen.doLogin()</td>
</tr>
<tr>
<td>01:25</td>
<td>$\Rightarrow$</td>
<td>ClientScreen.actionPerformed $\Rightarrow$ ClientScreen.doLogin()</td>
</tr>
</tbody>
</table>

Table 12.2: An example session transcript

<table>
<thead>
<tr>
<th>Participant</th>
<th>$\sim$</th>
<th>$\oplus$</th>
<th>$\ominus$</th>
<th>$\ominus$</th>
<th>$\Rightarrow$</th>
<th>$\Rightarrow$</th>
<th>$\Rightarrow$</th>
<th>$\Rightarrow$</th>
<th>$\Leftarrow$</th>
<th>$\Leftarrow$</th>
<th>$\Leftarrow$</th>
<th>$\Leftarrow$</th>
<th>$\Leftarrow$</th>
<th>$\Leftarrow$</th>
<th>$\Leftarrow$</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-T1-A1-E</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>23</td>
<td>31</td>
</tr>
<tr>
<td>P1-T1-A2-V</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P1-T2-A1-V</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>P1-T2-A2-E</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>19</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>46</td>
<td>60</td>
</tr>
</tbody>
</table>

Mean: 3.00, 0.00, 0.75, 0.00, 0.25, 0.00  
Median: 3.00, 0.00, 2.50, 4.50, 1.00  
Range: 3.00, 0.00, 8.00, 10.00, 6.00  

Table 12.3: An example of the navigations performed by a participant
Table 12.4: An example summary of the navigations performed by a participant

<table>
<thead>
<tr>
<th></th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-T1-A1-E</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>P1-T1-A2-V</td>
<td>0</td>
<td>1</td>
<td>18</td>
<td>10</td>
<td>29</td>
</tr>
<tr>
<td>P1-T2-A1-V</td>
<td>0</td>
<td>7</td>
<td>21</td>
<td>14</td>
<td>42</td>
</tr>
<tr>
<td>P1-T2-A2-E</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0</td>
<td>16</td>
<td>42</td>
<td>113</td>
<td>171</td>
</tr>
</tbody>
</table>

12.5 Experimental design

The experimental design is a within-subjects, counter-balanced design though the exact form of the design is specific to each study. A study is designed as either being a between-subjects design, where each participant only uses one condition (one development environment); or a within-subjects design, where each participant uses both conditions (both development environments). A between-subjects design is typically used where the difference between individuals is small and the number of individuals taking part in the study is large. In such a study, the intent is to statistically determine what the effect of each condition (or treatment) is on a large population of participants. Such a design is typically used in pharmaceutical studies where the aim is to reveal natural differences between people. In contrast, a within-subjects design is typically used where there is a wider difference between individuals and it becomes relevant to discover what the difference is for each individual person using each condition (each development environment). When evaluating software development tools, a within-subjects design should be used as this allows the analysis to compare and contrast how each participant uses each condition.

The learning effect is a key problem with studies using a within-subjects design – once a participant has completed a task using one condition, they remember what they have done when they attempt the task with the different condition. A strategy used in psychology is to create another task that is similar to the first task – in form and complexity – but requires a different solution. An example for a word memorisation test might be to have a second list of different words that are similar in length to the original list. Unfortunately, such a strategy is not easily applicable to programming tasks. For software navigation tasks, the structure and content of the codebase is what is most susceptible to the learning effect, however, if the codebase is changed, the task is almost certainly likely to be significantly changed. For software composition, similar programming tasks are easily generalisable – in other words, the solution to the second task would be easily recognised as being a variant on the solution to the first task. While the learning effect cannot be removed, it can be accounted for by using a counter-balanced experimental design where each participant group experiences a different treatment – a different combination of condition (development environment) and task. Another strategy to control for the learning effect is to include a training task so that the learning effect influences all tasks more equally than otherwise.
For the studies carried out during this project, it was decided not to use a training task because a goal of the study was to investigate how participants navigated an unfamiliar codebase. In hindsight, however, it may have been useful to carry out a training task on a different codebase solely to familiarise participants with Visuocode.

12.6 Analysis

The analysis method for the data collected involves quantifying navigation behaviour by coding programmer navigations, as well as qualitatively analysing each participant session to identify common strategies and issues. While statistical analysis becomes difficult when a small number of participants are involved in a study, some statistical tests can be performed to provide additional insight into the quantified results obtained. A repeated measures t-test can be used to analyse the average difference in the number of each kind of navigation each participant performed while using each experimental condition, i.e., development environment. It is emphasised that the statistical analysis is just one component of a qualitative analysis that studies the navigation behaviour of each individual participant.

12.7 Summary

This chapter described an experiment plan for analysing the software navigation behaviour of programmers. Experienced programmers are asked to attempt various programming tasks using both the Visuocode and Eclipse programming environments. Data is collected qualitatively through the use of screen-capture software that also records audio. Each participant session is coded into a session transcript that records the navigations performed. The navigations are analysed by identifying common strategies and issues, as well as by comparing the navigation behaviour of programmers with the coded navigations. The next two chapters describe how this experiment plan was used to compare the software navigation behaviour exhibited by participants while using a traditional development environment (Eclipse) and a prototype development environment that aims to provide better support for the integration of software navigation journeys into a cognitive map within long-term spatial memory (Visuocode). It is thought that better support for spatial memory will encourage more relative navigations than otherwise. The first study investigates programmer behaviour during software understanding motivated tasks (see Chapter 13), and the second study investigates programmer behaviour during program composition tasks (see Chapter 14).
Chapter 13

An investigation of software navigation

Studies have revealed that programmers spend a large proportion of their time navigating software [K⁺06]. Unfortunately, during such navigation, programmers often become disoriented and lose task awareness [dAM06]. It is thought that disorientation may be caused by the way that traditional development environments manage code editors, as when a programmer navigates to a different source code file, the existing editor is either replaced or obscured. Providing explicit support for relative navigation might mitigate programmer disorientation and increase programmer performance.

Several studies [KAM05, K⁺11, KKK⁺13] have revealed that programmers often use a two-phase strategy while exploring unfamiliar source code. First they carry out an exploration phase during which they opportunistically investigate source code files until they find a section of code that is potentially relevant – referred to as an anchor point. Next they begin a traversal phase during which they explore the call graph from the anchor point.

The method-flow software visualisation technique, which has been implemented within the Visuocode prototype development environment, explicitly supports relative software navigation by presenting source code within editor columns that are arranged within a horizontally scrollable flow view. If a hyperlink-enabled method call is followed, an editor column corresponding to the called method is placed adjacent to the calling column. It is thought that this should leverage visuo-spatial memory because each editor column is spatially separated within the flow view, mitigating disorientation as the programmer’s existing programming context is maintained.

Method-flow should provide the most support to programmers during the traversal phase while they are exploring the call graph. If method-flow mitigates disorientation and increases programmer performance, it is hypothesised that programmers should favour relative navigation more while using Visuocode than while using Eclipse. In particular, Visuocode should support those programmers who prefer to use a top-down strategy as such a strategy emphasises navigations that explore the call graph of a program. If relative navigation is more effective than direct navigation, it is expected that programmers will perform more relative navigations, and fewer direct navigations, while using Visuocode than while using Eclipse.
This chapter reports on a qualitative study that compares the navigation behaviour of programmers attempting software understanding tasks while using both Visuocode and Eclipse. Participants were asked to navigate and comprehend two pieces of software motivated by discovering how to carry out perfective changes. Screen-capture recordings of participant sessions were analysed to identify the type and number of navigations that each participant performed while using each environment. In addition, the behaviour of each participant was analysed to identify differences in navigation behaviour while using the different environments. The format of this chapter is adapted from the guidelines published by Jedlitschka [JCP08]. Section 13.1 describes the technologies under investigation, alternative technologies, and related studies. Section 13.2 describes the intended experiment plan. Section 13.3 describes deviations from the experiment plan. Section 13.4 summarises the participant session. Section 13.5 analyses the number and type of navigations performed. Section 13.6 analyses how participants used method-flow. Section 13.8 discusses the results of the post-study questionnaire. Section 13.9 collects the findings of the study. Section 13.10 discusses the findings. Section 13.11 presents the conclusions.

13.1 Background

13.1.1 Technologies under investigation

The technologies under investigation are the Eclipse IDE and the Visuocode prototype development environment. Visuocode has been describe previously in Chapter 11, but is described again to provide a comparison with Eclipse.

Visuocode

Visuocode is a prototype development environment for the Java programming language that implements the method-flow visualisation technique. It presents two different forms of window: the Workspace Manager window, and the Flow window, which contains a flow view (see Figure 13.1). The Workspace Manager facilitates direct navigation by presenting a tree structure that allows the programmer to reveal projects, packages, source code files, classes, and methods. Selecting a class or method empties the flow view and causes a corresponding editor column to be presented within the flow view, which is a horizontally scrollable view that can contain an essentially unlimited number of adjacently positioned editor columns. An editor column is a structured representation of the source code of a class. Each editor column contains a class declaration label, a list of imported packages and classes, a list of enumerations, a list of class members, and a scrollable methods area containing vertically stacked method editors. Within each method editor, class types and method calls are represented as hyperlinks. Following such a link causes an editor column to be shown adjacent, and to the right of, the editor column containing the followed link. If a method call link was followed, the editor column
Figure 13.1: The Visuocode Workspace Manager and Flow windows

only shows that one corresponding method; otherwise, if a class link was followed, editors for all of the class’s methods are presented within the scrollbar methods area. If more editor columns are contained within a flow view than can be displayed, the programmer may scroll back and forth horizontally along the call graph branch. Similarly, if multiple methods are represented within the methods area of an editor column, the programmer may scroll up and down vertically. Visuocode makes no attempt to recognise or prevent a programmer navigating down a recursive sequence of method calls. Due to each specific method being backed by a single model object, if the programmer decides to modify a method, any other representations of the same method are also be updated.

The build of Visuocode used for this study only allows one Flow window to be open at a time as allowing multiple windows would introduce an additional window management problem that it was thought best to avoid for this study. Typically, other development environments try to avoid window management by incorporating all views into one window and providing tabbing support within that window.

Eclipse

Eclipse is a popular open source software development environment for the Java programming language. Usually, Eclipse users work with a single window that is divided up into separate areas (panes) for auxiliary views and code editors. Figure 13.2 shows a typical Eclipse window. On the left-hand side of the window, the Package Explorer provides a tree structure representing a virtual representation of the file system including projects, packages, classes, and class members;
Figure 13.2: The Eclipse IDE

while below it the Outline view shows the members of the class within the active code editor. Taking up most of the window on the right, code editors are represented within a tabbed pane. Each code editor shows the content of a specific source code file. Typically, an Eclipse code editor presents code as it is represented within the source file unless a programmer has intentionally (or inadvertently) collapsed sections of code through code elision. Eclipse also supports dragging an open code editor about the Eclipse window, and dropping it into a different editor tab group (or out into a new window) – this makes it possible to drag an editor to the right-side of the window in order to display two editors side-by-side. Below the editors pane, there is usually another area that contains a variety of auxiliary views including the Problems view. The Eclipse environment is extremely extensible and customisable. Each auxiliary view may be dragged to another part of the window – either as a tab within a tab group, or to its own area.

Eclipse supports both direct and relative navigation. Direct navigation may be performed by selecting a reference in the Package Explorer, the Outline view, or many of the other auxiliary views. A forward relative navigation may be performed by hyperlink-enabling a method call by hovering the mouse pointer over it and holding down a modifier key. Following such a method call causes the corresponding code editor to appear – scrolling or obscuring the current editor. Additionally, the user may use a context menu to open a Call Hierarchy view that lists the methods that call the selected method and those methods that it calls. This allows a programmer to navigate up and down the call graph. A user may also navigate based on their navigation history by using the forward and back buttons. A limitation of Eclipse is that it
can only display one code editor at a time for a specific source code file. This means that if the user navigates to a method that is defined within the current editor, that code editor is scrolled to display the corresponding method, disrupting the existing programming context.

13.1.2 Alternative technologies

As this study focuses on software understanding, there are various software exploration tools (described in Section 4.3.4) that may be considered as alternative technologies. In particular, Rigi [MK88] and SHriMP [SBM+02] allow the user to see a top-level abstract view of software, and then drill down into specific source code files. Examples of mainstream software development environments that provide explicit support for Java are IntelliJ and NetBeans, while examples of recent research software development environments that seek to either support navigation, or introduce new navigation paradigms include Code Canvas [DR10], Code Bubbles [B+10], and Patchworks [HF14]. Tools that provide explicit support for relative navigation include: Stacksplorer [K+11], which shows columns on each side of the source code editor that contain navigable links to methods calling or called by the current method, respectively; Blaze [KKKB12], which shows an entire call graph branch in a view to the right of the source code editor – Blaze uses a combination lock metaphor that allows the user to select different possible branches by clicking left and right arrows presented for each method of the call graph; and Prodet [AFQ+15], which provides a search view, a navigation view, and a map view – selecting a suggestion from the search view causes the navigation view to show the call graph surrounding the selected method, and the map view represents classes that have been explored.

13.1.3 Related studies

Related studies of software navigation have often compared the use of a novel tool with that of an existing mainstream programming environment. Due to quantitative evaluations often not producing statistically significant results (e.g., [MR92, dAMR07, K+11]), more recent studies [dAMR07, KKK+13, HF14] have included a qualitative component that provides richer data for analysis, but which also takes longer to analyse, and limits the feasible number of participants, which impacts statistical significance.

Meyers and Reiss reported [MR92] on a between-subjects quantitative study that compared the performance of 43 novice students using the FIELD [Rei95] development environment to those that used the environment with novel auxiliary views, however, due to participants swapping groups, they were only able to obtain mixed results.

Storey et al. reported [SWM97] on a qualitative study of 30 graduate and senior undergraduate students who were asked to perform software understanding tasks on a software implementation of the Monopoly board game using three different software exploration tools – Rigi [MK88], SHriMP [SBM+02], and SNiFF+ [Mor95]. They found that participants used a variety of strategies to complete the set tasks including using prior knowledge to guide search.
queries (if the environment supported search), opportunistically searching for source files relevant to the task, and systematically searching by tracing execution. They also found that the “graphical subsystem hierarchy presented by Rigi and SHriMP was effective at conveying a mental map of the program” [SWM97, p. 18] to the participants, however, as this was pre-created using a manual editor, it may have misled the participants regarding the actual structure of the software. Importantly, they noted that the exploration functionality provided by the tools did complement the participants comprehension strategies even though at times participants were frustrated by the lack of other functionality such as textual search.

DeAlwis et al. [dAMR07] reported on a quantitative study of 18 professional programmers that investigated the effectiveness of three software exploration plug-ins for Eclipse. Each participant was asked to investigate and document a solution for two change tasks, and were given 40 minutes to complete each task. Participants were measured using self-reported solutions, and their navigation behaviour. Due to differences in tasks, however, they were unable to reliably find any significant difference between the participants that used Eclipse and those who used the other tools.

Karrer et al. [K+11] reported on a quantitative study of 16 participants that investigated how practitioners use Stacksplorer’s features while working with unknown code. Each participant was asked to solve two similarly difficult programming tasks, and were given 40 minutes for each task. Participants were measured using task performance and success rates. They were unable to find a significant difference for both tasks, however, the post session questionnaire did reveal that more than half the participants strongly agreed that navigation was faster using Stacksplorer.

Krämer et al. [KKK+13] reported on a between-subjects quantitative study of 33 participants that compared navigation behaviour while using Xcode 3 and three software navigation support tools – Call Hierarchy, Stacksplorer, and Blaze. Each participant was asked to attempt two tasks – they were given 25 minutes to complete Task 1, and 15 minutes to complete Task 2. They found that participants completed their tasks significantly faster if using the experimental conditions than if using the controls, but no difference between the individual controls. They found that participants performed better if using Stacksplorer and Blaze than if using the Call Hierarchy tool, and also that those using the three navigations tools performed better than those using plain Xcode. They concluded that the tools allowed the participants to explore the code using more efficient search strategies, however, they also noted that the task attempted had a significant effect on the accuracy of their prediction models.

Henley and Fleming [HF14] describe a study of 15 participants who used Eclipse, Code Bubbles, and Patchworks to perform navigation actions. For each environment, the participants were first asked to layout the content of 30 files that were considered relevant to a task, then were asked to navigate to specific methods. While they were able to produce results to support their specific research questions, the authors admit that the ecological validity of the tasks might be questioned – i.e., how their findings generalise to real life settings – due to only
providing the participants with a subset of the program codebase.

Relevant studies related to program composition are discussed in Section 14.1.3.

13.2 Experiment planning

13.2.1 Goal

The goal of the study was to investigate the difference in software navigation behaviour while participants explored unfamiliar source code using a mainstream development environment (Eclipse) and one that implements method-flow (Visuocode).

13.2.2 Participants

The study involved six participants: three academics and three PhD students. Each academic was considered an expert programmer – having taught one or more programming related subjects and/or being experienced with more than one programming language. Each student had multiple years experience in either Java or a suitably similar object-oriented programming language, such as C# – for more details regarding participants refer to Table 13.14. The academics selected were colleagues who had been known for several years including one advisor, and one of the students was a friend – the other two students were not known by the author before the study. The participants received no compensation for taking part in the experiment.

13.2.3 Experimental equipment and materials

Each participant was seated at a desk with a 24 inch monitor, keyboard, and mouse. These were connected to an Apple MacBook laptop computer. The version of Eclipse used was Eclipse Kepler, and the version of Visuocode used was 0.5.4 (configured to only have one flow view window, and with code modification disabled). Before beginning the experiment, each participant was shown a short training video for both Eclipse and Visuocode that highlighted their navigation capabilities. For example, participants were shown how to perform a relative navigation in Eclipse by holding down the Apple Command key and hovering the mouse over a method call. Use of ‘go to definition’ or the Call Hierarchy view was not discussed, but was relayed to participants during their session if they asked about such functionality. Before each task, the participant was also shown a short video that demonstrated the execution of the software whose source code they were about to explore. After the experiment, each participant was asked to complete a questionnaire that included a standardised System Usability Scale question sheet [Bro96], as well as questions regarding their programming history, preference of software development environment, and what features they might want implemented within Visuocode.
Source selection

In order to make the experiment as representative of real world programming as possible, the source code to be navigated for each task, was an existing, fully functional open source application. SourceForge was searched for appropriate open source Java applications. In order to encourage enthusiasm for the tasks, an initial shortlist was produced based on applications that might seem interesting for the participants to explore. This list was then reduced to only include software that was able to be easily downloaded, compiled, and run. The resulting list was then reduced again based on the suitability of the source code – the main consideration being that the application was Java self-complete, meaning that it did not rely on other dependencies that would not be navigable. Finally, TextEditor++ [Puz09] and JavaChat [Ser05] were chosen. Table 13.1 shows the versions of TextEditor++ and JavaChat that were used, as well as breakdown of the number of classes and the lines of code of each piece of software. TextEditor++ provides a good example of a small Java program that is contained within a single package, while JavaChat was chosen due to it being a more complex, client-server, multi-package, multi-project system. Software that had been used in previously published studies was also considered, however these were either considered too large and complex (JEdit [HF14]) or were not readily available (ShapeDraw [B+10]).

13.2.4 Tasks

Each participant was asked to attempt two tasks – each task required them to navigate program source code in order to determine how they might modify the code to add some functionality (see Table 13.2). The first task involved navigating the source code of TextEditor++, while the second task involved navigating the source code of JavaChat. Each task contained two similar activities – the participants were asked to attempt the first activity using one development environment, then the second activity using the alternate development environment. In contrast to the studies described previously (in Section 13.1.3) that either reduced the number of files involved in the study (Patchworks [HF14]) or suggested a specific method as a starting point (Code Bubbles [B+10]), participants were given no initial guidance regarding which files to navigate. Each participant was given 10 minutes to work on each activity. The study task sheets used are included in Appendix A.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Number of classes</th>
<th>Lines of code (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextEditor++</td>
<td>1.2.0</td>
<td>16</td>
<td>1796</td>
</tr>
<tr>
<td>JavaChat</td>
<td>1.1 (Snapshot)</td>
<td>63</td>
<td>2768</td>
</tr>
</tbody>
</table>

Table 13.1: Study 1: Size of task source code
13.2.5 Solutions

For each activity of each task, the target codebase was searched thoroughly to determine if the desired functionality already existed, or how it might be implemented. This led to an understanding of which classes are most important for the completion of each task – those that either needed to be modified to provide the desired functionality, or those that needed to be traversed in order to find where to implement the desired functionality. For Task 1, the most important classes of TextEditor++ are the MainFrame and TextEditorPane classes. The MainFrame class is directly referenced from the program entry point, and MainFrame directly references the TextEditorPane class, which provides all of the editor functionality. The TextEditorPane class inherits from the JTextArea class, which is provided by the Java standard library. A solution involves modifying the TextEditorPane to use methods provided by the JTextArea, and its parent classes, to colour selected sections of text. For Task 2, the most important classes of JavaChat are the ClientManager and ClientScreen classes. The ClientManager class is referenced directly from the program entry point, and instantiates a ClientScreen object as a member within its initialisation code. The ClientScreen class acts as a listener for numerous interfaces, and acts as the main controller of the program. In particular, it contains the doRegister and doLogin methods which are directly involved in the registration and login processes.

<table>
<thead>
<tr>
<th>Task 1 – TextEditor++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity 1</td>
</tr>
<tr>
<td>Activity 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task 2 – JavaChat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity 1</td>
</tr>
<tr>
<td>Activity 2</td>
</tr>
</tbody>
</table>

Table 13.2: Study 1: Description of activities
13.2.6 Hypotheses

It is thought that disorientation is caused by a lack of support for relative navigation in software development environments as it appears that programmers become disoriented when they realise they do not know how to retrace their steps after navigating through source code. This realisation then prompts them to perform a direct navigation to a known location [dAM06]. The method-flow visualisation technique, implemented within Visucode, aims to support visuospatial memory by providing time for egocentric information related to software navigation journeys, i.e., the method editors traversed, to be integrated into an allocentric spatial cognitive map within long-term memory. This should produce two effects that increase the number of relative navigations performed: firstly, the programmer feels that they have better user interface support for relative navigation because they can easily backtrack to previously traversed methods; secondly, the programmer is better able to remember how to return to previously visited methods that are near (on the call graph) to methods represented within editor columns contained in the flow view due to previous egocentric navigation information being integrated into an allocentric cognitive map of the software – this encourages the programmer to use relative navigation to return to the those locations.

Therefore, the research question of this study is related to whether the use of method-flow affects the navigation behaviour of participants. The research question is:

RQ1: which environment encourages more relative navigations?

The null hypothesis for this research question is that participants perform a similar number of relative navigations regardless of environment, or that participants perform fewer relative navigations while using Visucode:

\[ H_0 = \text{RELATIVE( ECLIPSE )} \geq \text{RELATIVE( VISUCODE )} \]

The alternative hypothesis is that participants perform more relative navigations while using Visucode:

\[ H_A = \text{RELATIVE( ECLIPSE )} < \text{RELATIVE( VISUCODE )} \]

13.2.7 Experimental design

The study has a within-subjects, counter-balanced experimental design – each participant used both conditions – Eclipse and Visucode. Half the participants (Group A) used Eclipse for the first activity of Task 1 and the second activity of Task 2. The other half (Group B) reversed this order for each task (see Table 13.3). Participants were allocated to each of these groups randomly. This design allows comparison of the usage of the two different environments by the same programmer. By reversing the activities for Task 2, the participants may be observed exploring an unfamiliar codebase for the first time using each environment. The dependent
variables were the number and type of navigations that participants performed while attempting the tasks, which were obtained by coding each participants navigations using the methodology discussed in Section 12.4.

### 13.2.8 Procedure

The procedure included four phases: Orientation, Preparation, Task Completion, and Post-study Questionnaire. During Orientation, each participant was seated at the computer desk and provided with an information sheet to read. It was then emphasised that the study was not intended to evaluate their programming ability but rather to investigate how they navigated software using the development environments provided. It was further highlighted that they need not complete the tasks, but that they were to consider the tasks as a hypothetical motivation for navigating the software. They were then required to sign a consent form. During Preparation, each participant was asked to watch a training video for each environment, which took approximately 5 minutes – Visuocode then Eclipse. During Task Completion, for each task, the participant was asked to read a task sheet that described the task and the two activities they would be attempting, then they were asked to watch a video that demonstrated the software. Each participant was then given 20 minutes to complete the two activities – 10 minutes with one environment and 10 minutes with the other. The participants were allowed to navigate the source code in any way they chose including performing global search, or local find. The only restriction was that they were asked not to set break points and run the program in debugging mode. The participants were given the option of a five minute break between each task. During the final phase, the participants were asked to complete a post-study questionnaire. First, they were asked to complete a standardised System Usability Scale [SL11], then they were asked to answer several questions related to their programming history, preference of development environment, and thoughts on Visuocode.

### 13.2.9 Data collection

During the completion of the tasks, each participant was recorded (video+audio) using an ‘over-the-shoulder’ angle video camera, and their computer interaction was recorded using

<table>
<thead>
<tr>
<th>Task</th>
<th>Activity</th>
<th>GROUP A</th>
<th>GROUP B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Activity 1</td>
<td>Eclipse</td>
<td>Visuocode</td>
</tr>
<tr>
<td></td>
<td>Activity 2</td>
<td>Visuocode</td>
<td>Eclipse</td>
</tr>
<tr>
<td>Task 2</td>
<td>Activity 1</td>
<td>Visuocode</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td>Activity 2</td>
<td>Eclipse</td>
<td>Visuocode</td>
</tr>
</tbody>
</table>

Table 13.3: Study 1: Experiment design – a within-subjects, counter-balanced design

Refer to Table 13.2 for activity descriptions
screen capture software that also recorded audio. The post-study questionnaire was completed electronically within two separate word processor files.

13.2.10 Analysis procedure

Analysis of the users sessions was performed using the approach described in Chapter 12.

13.3 Execution deviations

If the participant felt they had not yet completed the first activity to a satisfactory degree, they were given the option to continue that activity with the alternative development environment – the two activities for each task were intentionally similar enough that this should not have affected the results. After the completion of each activity, in order to maintain motivation, the experimenter debriefed the participant regarding their planned solution even though completion of the task was not strictly required.

13.4 Participant summary

The study involved six participants who are referred to as P1, P2, . . ., P6. The participants P1, P2, and P3 carried out the tasks in the order Eclipse-Visuocode-Visuocode-Eclipse (EVVE), while P4, P5, and P6 carried out the tasks in the order Visuocode-Eclipse-Eclipse-Visuocode (VEEV). Appendix B includes an analysis of each participant session.

Table 13.4 shows the strategies used by each participant during each task. While attempting Task 1, four of the participants used an opportunistic strategy, which is characterised by the participant clicking semi-randomly into files whose names appeared related to the task. In contrast, while attempting Task 2, all participants other than P6 used a top-down strategy. Where the strategy is described as ‘None’ the participant used no navigation strategy because they had already identified the classes relevant to the task. Interestingly, while using Eclipse during the last activity, both P2 and P3 used relative navigation noticeably more than during the first activity with Eclipse. One might surmise that experience with Visuocode’s method-flow

<table>
<thead>
<tr>
<th>Task 1</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity 1</td>
<td>Top-down</td>
<td>Opportunistic</td>
<td>Opportunistic</td>
<td>Opportunistic</td>
<td>Top-down</td>
<td>Opportunistic</td>
</tr>
<tr>
<td>Activity 2</td>
<td>Top-down</td>
<td>Top-down</td>
<td>None</td>
<td>Top-down</td>
<td>Top-down</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task 2</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity 1</td>
<td>Top-down</td>
<td>Top-down</td>
<td>Top-down</td>
<td>Top-down</td>
<td>Top-down</td>
<td>Opportunistic</td>
</tr>
<tr>
<td>Activity 2</td>
<td>Top-down</td>
<td>Top-down</td>
<td>Top-down</td>
<td>None</td>
<td>Top-down</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 13.4: Study 1: Navigation strategies used by participants
navigation may have affected their navigation style, however, it is also possible this difference may simply be due to the difference in task and codebase size.

As Visuocode explicitly supports navigating down call graph branches it was expected that programmers would navigate deeper using Visuocode than while using Eclipse. Analysis of each participant’s navigations revealed that one participant (P1) navigated deepest while using Visuocode, three participants (P2, P3 & P6) navigated deepest while using Eclipse, and the remaining two participants (P4 & P5) navigated to the same depth using both environments. After averaging the depths, three participants (P2, P5, & P6) navigated deepest using Eclipse, while only two participants (P1 & P4) navigated deepest using Visuocode. The remaining participant (P3) navigated to an average depth of 3 methods using both Eclipse and Visuocode, but navigated deepest using Eclipse.

Of the participants, P5 performed the most number of scrolling navigations while using Eclipse during Task 1 Activity 2. All of these 94 navigations occurred within the `MainFrame` and `TextEditorPane` classes due to first opening one of the classes, and scrolling through it, before opening the other class, and scrolling through it. In contrast, while navigating the same codebase using Visuocode during Activity 1, P5 only performed 39 scrolling navigations – this was due to adopting the navigation style of opening a single method from the Workspace Manager and navigating horizontally using the method-flow instead of vertically by scrolling. In particular, they were able to study the relationship between the `openDocument` and `addDocument` methods of `MainFrame` by having them juxtaposed within a flow view. P1 performed the second highest number (58) of scrolling navigations while using Eclipse during Task 1 Activity 1. This was due to first scrolling through the `MainFrame` class then scrolling through the `TextEditorPane` class. Similarly, while using Eclipse during Task 2 Activity 2, P1 performed numerous scrolling navigations to navigate through the `ClientManager` and `ClientScreen` classes. In contrast, while using Visuocode, P1 used method-flow to extensively explore the call graph

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th></th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td></td>
<td></td>
<td></td>
<td>Task 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity 1 (E)</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>Activity 1 (V)</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Activity 2 (V)</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>Activity 2 (E)</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Task 2</td>
<td></td>
<td></td>
<td></td>
<td>Task 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity 1 (V)</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>Activity 1 (E)</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Activity 2 (E)</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>Activity 2 (V)</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Avg. Eclipse</td>
<td>2.5</td>
<td>4</td>
<td>3</td>
<td></td>
<td>3</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>Avg. Visuocode</td>
<td>3</td>
<td>3.5</td>
<td>3</td>
<td></td>
<td>3.5</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>Deepest.</td>
<td>V</td>
<td>E</td>
<td>E</td>
<td></td>
<td>=</td>
<td>=</td>
<td>V</td>
</tr>
<tr>
<td>Avg. Deepest.</td>
<td>V</td>
<td>E</td>
<td>=</td>
<td></td>
<td>V</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 13.5: Study 1: Maximum navigation depths
around the *ClientScreen.start* method instead of scrolling through files.

### 13.5 Quantitative analysis

The quantitative analysis consisted of coding the navigations that were performed by each individual participant, and then analysing whether the number of each type of navigation was dependent on either the participant, task, or environment. Additionally, three matching samples t-tests were performed to analyse the difference between the number of direct, relative, and scrolling navigations performed by each participant. Section 13.5.1 compares the navigations performed by each participant. Section 13.5.2 compares the navigations that were performed during each task. Section 13.5.3 compares the navigations that were performed while using each environment. Section 13.5.4 presents the results of a statistical analysis comparing direct, relative, and scrolling navigations performed. Section 13.5.5 summarises the results.

#### 13.5.1 Results by participant

Table 13.6 shows the navigations grouped by kind. Except for P4, all participants predominantly performed scrolling navigations during their activities. P1 and P2 performed more relative navigations than direct navigations, while the other participants performed more direct navigations than relative. Table 13.7 shows the total number of each navigation type performed by each participant. Of the direct navigation types, all participants predominantly performed file navigations. Of the relative navigation types, participants predominantly performed relative navigations without using the Eclipse Call Hierarchy view. All participants performed tabbing navigations during at least one Eclipse activity.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>16</td>
<td>42</td>
<td>113</td>
<td>171</td>
</tr>
<tr>
<td>P2</td>
<td>45</td>
<td>49</td>
<td>77</td>
<td>171</td>
</tr>
<tr>
<td>P3</td>
<td>22</td>
<td>13</td>
<td>62</td>
<td>97</td>
</tr>
<tr>
<td>P4</td>
<td>45</td>
<td>27</td>
<td>15</td>
<td>87</td>
</tr>
<tr>
<td>P5</td>
<td>72</td>
<td>43</td>
<td>210</td>
<td>325</td>
</tr>
<tr>
<td>P6</td>
<td>72</td>
<td>60</td>
<td>82</td>
<td>214</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>272</strong></td>
<td><strong>234</strong></td>
<td><strong>559</strong></td>
<td><strong>1065</strong></td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>45.33</td>
<td>39.00</td>
<td>93.17</td>
<td>177.50</td>
</tr>
<tr>
<td><strong>Std Dev.</strong></td>
<td>23.78</td>
<td>16.65</td>
<td>65.60</td>
<td>87</td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td>45</td>
<td>42.5</td>
<td>79.5</td>
<td>171</td>
</tr>
<tr>
<td><strong>Range</strong></td>
<td>56</td>
<td>60</td>
<td>195</td>
<td>238</td>
</tr>
</tbody>
</table>

Table 13.6: Study 1: Summary of navigations grouped by participant
### 13.5.2 Results by task

Table 13.8 summarises the navigations performed during each task. Unexpectedly, in total, more navigations were performed while attempting the TextEdit++ task than while attempting the JavaChat task. For both tasks, scrolling navigations were predominantly performed, though the number during the TextEditor task was nearly double the number performed during the JavaChat task. The number of direct navigations performed while attempting the TextEditor task was consistently higher than while attempting the JavaChat task, while the number of relative navigations performed while attempting the JavaChat tasks was consistently higher than while attempting the TextEditor++ task. Table 13.9 compares the number of navigations performed in each task.

<table>
<thead>
<tr>
<th>Task</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextEdit+++</td>
<td>145</td>
<td>62</td>
<td>369</td>
<td>576</td>
</tr>
<tr>
<td>JavaChat</td>
<td>127</td>
<td>172</td>
<td>190</td>
<td>489</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>272</td>
<td>234</td>
<td>559</td>
<td>1065</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>136</td>
<td>117</td>
<td>279.5</td>
<td>532.5</td>
</tr>
<tr>
<td><strong>Std. Dev.</strong></td>
<td>12.728</td>
<td>77.782</td>
<td>126.572</td>
<td>61.52</td>
</tr>
<tr>
<td><strong>Range</strong></td>
<td>18</td>
<td>110</td>
<td>179</td>
<td>87</td>
</tr>
</tbody>
</table>

Table 13.9: Study 1: Navigations performed during each task

### 13.5.3 Results by environment

Table 13.10 compares the number of navigations performed using each software development environment grouped by category. In total, participants performed many more (628 vs 437) navigations while using Eclipse than while using Visuocode, and this was also true of most individual activities. Furthermore, participants performed in total more than three times (179 vs
the number of relative navigations while using Visuocode than while using Eclipse. In particular, the number of scrolling navigations performed while using Eclipse is nearly three times the number performed while using Visuocode (417 vs 142). While using Eclipse, participants predominantly used scrolling navigations, then direct navigations, then relative navigations, whereas, participants predominantly used relative navigations, then scrolling navigations, then direct navigations while using Visuocode. Table 13.11 compares the number of navigations performed using each software development environment.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>156</td>
<td>55</td>
<td>417</td>
<td>628</td>
</tr>
<tr>
<td>Visuocode</td>
<td>116</td>
<td>179</td>
<td>142</td>
<td>437</td>
</tr>
<tr>
<td>Total</td>
<td>272</td>
<td>234</td>
<td>559</td>
<td>1065</td>
</tr>
<tr>
<td>Mean</td>
<td>136</td>
<td>117</td>
<td>279.5</td>
<td>532.5</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>28.284</td>
<td>87.681</td>
<td>194.454</td>
<td>135.06</td>
</tr>
<tr>
<td>Range</td>
<td>40</td>
<td>124</td>
<td>275</td>
<td>191</td>
</tr>
</tbody>
</table>

Table 13.10: Study 1: Summary of navigations performed using each environment

A repeated measures t-test can be used to investigate the average difference in the number of each kind of navigation performed by each individual participant while using each development environment. Three tests were performed to identify whether the use of a particular environment was associated with a difference in direct, relative, and scrolling navigations. The hypotheses for the three tests are the same except for the type of navigation under analysis. The null hypothesis ($H_0$) is that the mean of the Eclipse and Visuocode navigation differences is equal to zero, while the alternative hypothesis ($H_a$) is that the mean is not equal to zero. Since all three tests are repeated measures tests with $n = 6$, a two-tailed t-distribution is used with degrees-of-freedom = 5. Using an $\alpha = 0.05$, the null hypothesis is rejected if the test statistic is less than -2.571 or greater than 2.571.

Table 13.12 shows the results of the repeated measures t-tests. Participants performed significantly more relative navigations while using Visuocode than while using Eclipse (mean difference = 20.667, p-value = 0.0257), and participants performed significantly fewer scrolling navigations while using Visuocode than while using Eclipse (mean difference = -45.833, p-value = 0.0038). While participants performed fewer direct navigations while using Visuocode than while using Eclipse, the difference was not more than what can be explained by change alone.

13.5.4 Results of the repeated measures t-tests

A repeated measures t-test can be used to investigate the average difference in the number of each kind of navigation performed by each individual participant while using each development environment. Three tests were performed to identify whether the use of a particular environment was associated with a difference in direct, relative, and scrolling navigations. The hypotheses for the three tests are the same except for the type of navigation under analysis. The null hypothesis ($H_0$) is that the mean of the Eclipse and Visuocode navigation differences is equal to zero, while the alternative hypothesis ($H_a$) is that the mean is not equal to zero. Since all three tests are repeated measures tests with $n = 6$, a two-tailed t-distribution is used with degrees-of-freedom = 5. Using an $\alpha = 0.05$, the null hypothesis is rejected if the test statistic is less than -2.571 or greater than 2.571.

Table 13.12 shows the results of the repeated measures t-tests. Participants performed significantly more relative navigations while using Visuocode than while using Eclipse (mean difference = 20.667, p-value = 0.0257), and participants performed significantly fewer scrolling navigations while using Visuocode than while using Eclipse (mean difference = -45.833, p-value = 0.0038). While participants performed fewer direct navigations while using Visuocode than while using Eclipse, the difference was not more than what can be explained by change alone.
<table>
<thead>
<tr>
<th>Navigation kind</th>
<th>Mean of differences</th>
<th>Std Dev</th>
<th>t-Test score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>-6.667</td>
<td>14.306</td>
<td>-1.141</td>
<td>0.3053</td>
</tr>
<tr>
<td>Relative</td>
<td>20.667</td>
<td>16.113</td>
<td>3.138</td>
<td>0.0257</td>
</tr>
<tr>
<td>Scrolling</td>
<td>-45.833</td>
<td>22.158</td>
<td>-5.067</td>
<td>0.0038</td>
</tr>
</tbody>
</table>

Table 13.12: Study 1: Summary of results from repeated measures t-test

Visuocode vs Eclipse

13.5.5 Summary of results

Repeated measures t-tests that compared the number of direct, relative, and scrolling navigations revealed participants performed significantly more relative navigations and significantly fewer scrolling navigations while using Visuocode than while using Eclipse. A similar test of direct navigations revealed no difference other than what can be explained by chance alone.

13.6 Qualitative analysis of Visuocode (only)

Participant sessions that involved Visuocode were also analysed to understand how method-flow was being used – a description for each participant is provided in Appendix B.2. All participants used method-flow in a similar manner. Participants began their sessions by exploring the code project structure using the Workspace Manager to find a suitable anchor point. Once an anchor point was found, the participants used method-flow to explore the relationships between classes by exploring the local call graph. At times, when a new class was encountered, the participants would look for it in the Workspace Manager, then expand the class to see what methods it contained.

P6 performed the most relative navigations of any participant during Activity 2 of Task 2 while using the Visuocode environment – they performed 41 relative navigations and only 1 direct navigation. The navigations related to the call graph of the `ClientScreen.actionPerformed` method. In contrast, navigations during the previous activities had entailed a mix of scrolling, tabbing, call hierarchy, find, outline, and history navigations. P2 performed the second highest number of of relative navigations (24) during Activity 1 of Task 2 while using the Visuocode programming environment. During this activity, they performed zero scrolling navigations and only 13 direct navigations. In particular, for nearly half of this activity, the participant’s flow view contained four editor columns – this occurred due to extensive exploration of the call graph. In contrast, while using Eclipse, during Task 1 of Activity 1, P2 tended to use direct navigations to open files, scrolling navigations to scroll through those files, and tabbing navigations to tab between those files. Interestingly, during Task 2 Activity 2, after having used Visuocode for two activities in a row, P2 began to perform many more relative navigations within Eclipse.
13.7 Observations of disorientation

Disorientation is difficult to identify because it can manifest in many different types of behaviour. Additionally, no proposed guidelines for identifying disorientation were found in the studies reviewed in Section 13.1.3. To identify occasions of disorientation, the screen recordings of each participant session were reviewed again. Immediately, it became obvious that it is very hard to identify disorientation if participants are exploring an unfamiliar codebase because they are effectively lost to begin with. Four of the six participants showed some signs of disorientation, however, in some cases whether they were in fact disoriented is subjective. The screen recordings included audio, which proved very helpful to understand the programmers’ cognitive state. The disorientation that was observed either related to the programmer forgetting what they were searching for when navigating to a new class or losing track of where they had been when back-tracking.

P1 appeared to become disoriented twice. While attempting the Text Editor task using Eclipse, after navigating to the `MainFrame` class and scrolling down through the file looking for the `addDocument` method, P1 became confused regarding which method they were looking for after browsing past the `insertNewDocument` method. The second time related to becoming confused as to why the `TextEditorPane` did not include an expected method that was actually implemented in an ancestor class.

P4 showed the most occurrences of disorientation while navigating relatively using Eclipse, however, these were quite brief. First, while attempting the JavaChat task, P4 navigated from the `ClientScreen.doRegister` method to the `ClientScreen.setStatus` method; when they chose to return they first nearly selected the adjacent tab to the left before realising it was the wrong file and selecting the back button. Similarly, P4 later navigated from the `ClientScreen.doRegister` method to the `Dialogs.registerDialog` method, which caused a new editor to open that was beside a ‘Register.java’ tab they had previously opened; when they chose to return they clicked on the ‘Register.java’ tab and were momentarily confused about what they saw – they were unsure why that editor was there; rather than select the next tab to the left, which was the tab they wanted, they hit the back button twice to return to their initial context. Later, intending to navigate to the `ClientScreen.doRegister` method, P4 mistakenly navigated (via the Project Explorer) to the ‘Register.java’ file, but they quickly realised their mistake. Finally, while navigating the same codebase using Visuocode, during investigation of the initialisation of the `ClientScreen.screenCallback` member, P4 said “Oh I forgot where I got to that from...it was client screen dot start” – whether this is disorientation is debatable because they quickly remembered where they wanted to navigate to.

P5 only showed a couple of instances of what may loosely be regarded as disorientation – they are included here for completeness. While using Eclipse, after closing all their tabs, P5 then mistakenly opened the ‘TextEditor.java’ file (which only contains main) before realising their mistake and opening ‘TextEditorPane.java’. A little later they performed a direct navigation
to an already open ‘TextEditorPane.java’ file – this reflects more a different style of navigation rather than disorientation.

The other participants showed no identifiable signs of disorientation. Of note, however, was that P2 was confused when finding that the representation of the KeyTextListener.controlPress method was different in Visuocode than in Eclipse – investigation revealed that a previous participant had auto formatted what was a very long conditional that excluded certain key codes while using Eclipse.

In summary, the disorientation that was observed seemed related to two causes. First, the participant would perform a direct navigation to a class but either navigated to the wrong class or forget which method they intended to actually investigate as they scrolled down through the class. Second, while using Eclipse, the participant would perform a relative navigation that would have an unanticipated affect on the tabs or the editor making backtracking difficult – if the method navigated to was in the same class, the current source file would be scrolled but the participant would try to backtrack using the tabs; if the method navigated to was in a different class, the tab that was opened may not be adjacent to the previous class.

13.8 Post-study questionnaire

System Usability Scale

After completing the tasks, each participant was asked to fill out a standardised System Usability Scale questionnaire [Bro96]. Each question was answered on a Likert scale from 1 (Strongly disagree) to 5 (Strongly agree). Table 13.13 shows the participants’ responses, as well as the mean and standard deviations for each question – the questions have been reordered so that those with the highest average score are toward the top. From the responses, all participants found Visuocode simple to use and thought they would be able to use it without technical assistance (mean 4.33). Most participants found the Visuocode interface consistent, and did not think they would need to learn anything new to use the environment (mean 4.17). Visuocode scored less well with regards to integration (mean 4.00). This is probably due to the lack of integrated compilation and error detection, which is available in most mainstream environments. Participants opinions differed on whether most people would learn to use Visuocode quickly, or would find Visuocode intuitive – both these questions received an average score of 3.83 that resulted from responses containing both 5s and 2s. Similarly, participants were unsure whether they felt confident using Visuocode or whether Visuocode was easy to use. As might be expected from a prototype environment, Visuocode received the lowest average score (3.17) regarding whether the participants would use Visuocode frequently.
<table>
<thead>
<tr>
<th>Question</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>Mean</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>I found Visuocode to be simple</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.33</td>
<td>0.52</td>
</tr>
<tr>
<td>I think I could use Visuocode without the support of a technical person</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4.33</td>
<td>0.52</td>
</tr>
<tr>
<td>I thought there was a lot of consistency in Visuocode</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4.17</td>
<td>1.17</td>
</tr>
<tr>
<td>I could use Visuocode without having to learn anything new</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4.17</td>
<td>1.17</td>
</tr>
<tr>
<td>I found the various functions in Visuocode well integrated</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4.00</td>
<td>0.89</td>
</tr>
<tr>
<td>I would imagine that most people would learn to use Visuocode very quickly</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3.83</td>
<td>1.17</td>
</tr>
<tr>
<td>I found Visuocode very intuitive</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3.83</td>
<td>0.98</td>
</tr>
<tr>
<td>I felt very confident using Visuocode</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3.83</td>
<td>0.82</td>
</tr>
<tr>
<td>I thought Visuocode was easy to use</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3.33</td>
<td>1.03</td>
</tr>
<tr>
<td>I think that I would use Visuocode frequently</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3.17</td>
<td>0.98</td>
</tr>
<tr>
<td>Question</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
<td>P5</td>
<td>P6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency of use</td>
<td>one or two</td>
<td>one or two</td>
<td>infrequently</td>
<td>frequently</td>
<td>infrequently</td>
<td>infrequently</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experience w similar language</td>
<td>Python</td>
<td>C++</td>
<td>C Sharp</td>
<td>Python/Ruby/Obj-C</td>
<td>C++</td>
<td>PHP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency of use</td>
<td>very often</td>
<td>not in years</td>
<td>very often</td>
<td>now and then</td>
<td>not in years</td>
<td>addict</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-oriented (OO) exp.</td>
<td>pretty good</td>
<td>excellent</td>
<td>reasonable</td>
<td>pretty good</td>
<td>reasonable</td>
<td>excellent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preferred paradigm</td>
<td>undecided</td>
<td>functional</td>
<td>OO</td>
<td>undecided</td>
<td>OO</td>
<td>OO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preferred environment</td>
<td>Emacs</td>
<td>Vim</td>
<td>VS2013, Netbeans</td>
<td>Eclipse</td>
<td>Eclipse</td>
<td>Sublime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requested features</td>
<td>Search</td>
<td>Search</td>
<td>Search</td>
<td>Eclipse plug-in</td>
<td>Search</td>
<td>Comments</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Haskell support</td>
<td>Debugging</td>
<td>HLE Members</td>
<td>Comments</td>
<td>Search</td>
<td>Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Multiple FWs</td>
<td>Reverse nav.</td>
<td>Collapsed methods</td>
<td>Auto formatting</td>
<td>Popup doc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flow history</td>
<td>Cust. highlighting</td>
<td>Enhanced. Javadoc</td>
<td>Inc. compilation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WYSIWYG support</td>
<td>Search</td>
<td>Inc. compilation</td>
<td>Super class nav.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If requested features were implemented, would you use Visuocode</td>
<td>Yes</td>
<td>Unlikely</td>
<td>Possibly</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Free-form feedback

Table 13.14 shows a summary of the responses to the free-form questions. The point of these questions was to determine if any of the participants lacked appropriate experience, and to retrieve free-form feedback regarding the Visuocode environment. Although not every participant had recent Java experience, each participant did have extensive past or present experience with an object-oriented programming language – these languages included C++, C Sharp, Objective-C, Python, PHP, or Ruby.

In the free form part of the questionnaire, participants were asked what features they would like added to Visuocode. As can been seen from Table 13.14, the most requested feature was textual search (6 respondents). The next most requested feature was additional support for comments (3 respondents) – as these were stripped from methods. Some requests included extending the method-flow functionality, including making class members hyperlink-enabled, reverse navigation (1 respondent), collapsed/collapsible methods, allowing multiple flow windows, providing a history of traversed method-flows, and allowing superclass navigation. The remainder of the features requested were related to adding functionality that existing development environments already provide, such as incremental compilation and debugging support, enhanced Javadoc support, auto-formatting of text, tab support, and WYSIWYG (What You See Is What You Get) support for building mobile user interfaces for Android (such as is provided in Eclipse).

Each participant was asked, hypothetically, if this functionality were added to Visuocode, would they consider using the environment. Only one participant indicated ‘Yes’ while the rest were either unsure (probably trying to be polite) or indicated ‘No’. These responses indicate how difficult it is to have people migrate from the software development environment they know, and perhaps more importantly, trust.

13.9 Findings

The research question of this study was: “which environment encourages more relative navigations?” The null hypothesis that participants would perform a similar number of relative navigations or fewer while using Visuocode than while using Eclipse has been found not to hold, and the alternative hypothesis that participants would perform more relative navigations while using Visuocode has been found to be supported. Unexpectedly, the results also show that participants performed many more scrolling navigations while using Eclipse than while using Visuocode, and that the software to be navigated has an effect on the navigation strategy used, as well as the number and type of navigations performed. In summary, the findings of this study are:

1. Visuocode encouraged relative navigation
2. Eclipse encouraged scrolling navigation

3. The tasks affect navigation strategy

### 13.9.1 Visuocode encouraged relative navigation

It is found that Visuocode encourages more relative navigation than Eclipse because, in total, participants performed many more relative navigations while using Visuocode (179 vs 55) compared to while using Eclipse (refer to Table 13.10). When analysed by participant, every participant except P3 performed more relative navigations while using Visuocode than while using Eclipse. This finding is also supported by the result of a repeated measures t-test, which found that participants performed significantly more relative navigations while they were using Visuocode (see Section 13.5.4).

### 13.9.2 Eclipse encouraged scrolling navigations

It is found that Eclipse encourages more scrolling navigations than Visuocode because, in total, participants performed nearly three times as many scrolling navigations (417 vs 142) while using Eclipse than while using Visuocode (refer to Table 13.10). When analysed by participant, every participant performed more scrolling navigations while using Eclipse compared to while using Visuocode. This finding is also supported by the result of a repeated measures t-test, which found that participants performed significantly more scrolling navigations while they were using Eclipse (see Section 13.5.4).

### 13.9.3 Task affects navigation strategy

It is found that the tasks affected the number and type of navigations performed. While participants performed a similar number of direct navigations during each task (145 vs 127), in total, participants performed nearly double the number of file navigations (118 vs 66) while attempting the TextEdit++ activities than while attempting the JavaChat activities (refer to Table 13.8). Additionally, participants performed nearly double the number of scrolling navigations while navigating TextEdit++ than while navigating JavaChat (369 vs 190).

### 13.10 Discussion

The finding that participants performed more relative navigations while using Visuocode than while using Eclipse indicates that, in certain circumstances, programmers favour relative navigation if it is explicitly supported by the environment, and that Visuocode provides enough support for relative navigation that participants altered their navigation style while using the environment. Existing studies have observed that programmers often use a two-phase strategy while searching an unfamiliar codebase [KAM05, K+11, KKK+13]. First, they carry out
an opportunistic discovery phase during which they identify sections of code possibly relevant to their task, then they carry out a structured traversal phase during which they investigate the call graph from those sections of code. This study also observed such behaviour and that method-flow visualisation is most useful during the traversal phase. It was observed that when performing relative navigations while using Eclipse that participants would often need to either use a direct navigation to return to a previous known location, or would resort to using the Eclipse back navigation button to return to where they had been. It seems that Eclipse does not provide enough contextual information regarding the methods that a programmer has navigated through, and often editors are scrolled to other locations due to Eclipse only being able to have a single editor open for each source code file, which interferes with the strategy of using sequential editor tabs to remain oriented. This discourages programmers from using relative navigation and encourages the use of direct or scrolling navigations, which rely on the programmer's own mental model of the software due to their explicit nature.

Even though, intuitively, programmers should need to scroll less if navigating in a relative manner along the call graph, the extent of the difference exhibited between the two environments was surprising. While using Visuocode, participants would use the Workspace Manager to perform direct navigations to identify possibly relevant sections of code, then would use method-flow to better understand how these sections of code interacted with other classes. In contrast, while using Eclipse, participants would often navigate to a file then scroll down through it to judge its size and complexity, as well as how it interacted with other classes. While using Eclipse, participants seemed more focused on the composition hierarchy of classes whereas while using Visuocode, participants were more focused on the call graph.

Similar to findings by DeAlwis et al. [dAMR07], it is thought that both the task size and the form of the task affected the way that participants navigated the software codebases. Unexpectedly, participants performed more navigations in total while navigating the smaller TextEditor++ codebase than while navigating the larger JavaChat codebase. In addition, while investigating the TextEditor++ codebase, participants used a mix of opportunistic and top-down navigation strategies, however, while investigating the JavaChat codebase all participants but one used a top-down strategy. It is acknowledged that the task description may have affected navigation behaviour because the wording of the activities requested that the participants investigate program behaviour during program startup.

While using Visuocode, several of the participants were misled when a navigation was performed that resulted in only one method editor being presented within an editor column. For P1, this caused an expression of surprise when they later (while using Eclipse) found methods they had previously missed. P2 expressed at the end of their session that they found it strange swapping from a mind-set of seeing whole source code files. The term ‘blinkered’ is used to describe this disorientation that is produced during navigation when participants are not shown all the methods of a class. It is possible that there may be inherent issues with systems that only present single methods to the programmer, however, this may be exacerbated...
in Visuocode because each editor column shows all other class information including class declaration, imports, enums, and members. Additionally, all of a class’s methods are shown if a class link is followed while using Visuocode, which may increase the confusion caused if only one method is shown.

It is thought that the difference in navigation behaviour exhibited by participants supports the claim that method-flow supports the integration of egocentric navigation into an allocentric cognitive map within long-term spatial memory. As previously mentioned, P1 indicated surprise at finding methods that they had not navigated past during the previous activity, which indicates that the methods they had navigated through were integrated into some form of mental model.

13.10.1 Threats to validity

Threats to validity may include threats to: construct validity, which relate to whether the measures taken during an experiment are as meaningful as supposed in the real world; internal validity, which relate to the extent the treatments were responsible for the effects observed; external validity, which relate to how findings of the study may be generalised to other participant populations; and conclusion validity, which relate to whether statistical tests, upon which conclusions have been based, have been carried out correctly [JCP08].

Construct validity

For this study, the threat to construct validity relates to what degree software navigation behaviour actually affects productivity. It is an assumption that if a programmer performs fewer navigations than otherwise that they are able to spend any time saved comprehending or composing code, which should be reflected in increase programmer productivity. However, paradoxically, it is also assumed that if a programmer performs more navigations of a particular type then that type of navigation is favoured because it is more productive than other navigation types. It is also assumed that if a programmer shows fewer signs of disorientation while navigating that their efforts are likely to be more productive. It is emphasised that while these assumptions may be correct, there is currently no specific theory that rigorously supports it.

Internal validity

For this study, the threat to internal validity relates to the extent that the development environments used were responsible for the differences in navigation patterns observed. The main threat to internal validity is the affect of both the individual participants and the tasks on the results. Another potential threat to internal validity is the novelty of the experimental condition involved. It is possible that the participants may ‘try out’ navigating the software
in the style that is explicitly supported by the tool, and similarly change their usual style of navigation while using the control condition.

**External validity**

For this study, the two main threats to external validity are the artificial nature of the tasks attempted and the motivation of participants. The tasks are only relevant to a real world circumstance where a programmer has been tasked with the investigation of a codebase that is both unfamiliar to them and that they are unlikely to need to refer to it again in the future. This is likely to lead to participants being focused on finishing the task rather than building a mental model of the software – this was observed with both P3 and P6. To mitigate this tendency, during the Preparation phase of the study, it was emphasised that activities were intended to provide a motivation for navigating the software. However, it might be argued that the scenario described above – of navigating unfamiliar code – may actually have more real world relevance during software maintenance than it appears, as often new staff members are tasked with modifying software they are unfamiliar with, or have forgotten.

**Conclusion validity**

For this study, the conclusion validity relates to the confidence that the repeated measures t-tests were carried out correctly and are applicable for analysis of a problem of this sort. However, it is stressed that the statistics performed for this study have mainly been used to support the qualitative analysis, which mainly contributed toward the conclusions.

**13.10.2 Lessons learned**

For most participants, the non-prompted think-aloud protocol worked well as often the presence of the experimenter alone would prompt the participants to adequately describe what they were doing. Half of the participants became confused as to which activity they were supposed to be attempting first, i.e., they prematurely began the second activity – this did not affect the study because the activities were purposefully similar; participants were given the option to continue the first activity, if they had not yet completed it, when swapping to the next environment. For future studies, each activity will be provided separately. Two of the participants were very goal-focused during the tasks as they focused on completing the activities rather than navigating and understanding the software. In the future, the wording used on the task sheets as well as the activities chosen will be reconsidered to de-emphasis completing the tasks.

**13.11 Conclusions**

It is concluded that the method-flow visualisation technique implemented by Visuocode provides better support for relative navigation than the Eclipse IDE – which is especially important
during phases of programming that involve traversal of the call graph. Participants performed significantly more relative navigations while using Visuocode than while using Eclipse. This resulted from them being more effective during periods of relative navigation because maintaining editor columns within the method-flow prevented disorientation during backtracking. While using Visuocode, participants tended to explore the call graph of the codebase, whereas, in contrast, while using Eclipse, participants tended to explore the composition hierarchy by navigating directly to classes and scrolling down through them. While it is concluded that exploring the call graph using method-flow facilitates the integration of a programmer’s egocentric navigation into an allocentric cognitive map of the software, more research is needed to understand to what degree the resulting mental model differs from one built up in the traditional manner while using Eclipse. For example, while using method-flow, participants would be misled – ‘blinkerred’ – by only being shown a single method of a class instead of all of its methods.
Chapter 14

An investigation of program composition

There is both anecdotal and empirical evidence [KAM05] that programmers become disoriented while navigating through source code during software understanding tasks. Less is known about the navigation behaviour of programmers during program creation tasks – tasks where programmers create new software structure, rather than modify an existing operational software codebase. While traditional mainstream development environments provide features that are intended to aid software navigation and creation, there is little empirical evidence of how such environments are used.

An important activity of software maintenance is refactoring, which involves the identification of parts that need to be restructured, then extracting functionality into new classes or methods while leaving the functionality unchanged. Two attributes that may indicate code needs to be restructured are class size and method size [BFB99]. While, such refactoring may often be due to common code being intentionally factored out, it is possible that methods are large because the source code was not structured appropriately when first written. If programmers improperly include code because of a lack of explicit support for visually juxtaposing related methods, resulting in an increase of method size, then conversely, additional support should reduce the size of methods while increasing the number.

The Visuocode prototype development environment implements the method-flow visualisation technique, which allows a programmer to open horizontally adjacent editor columns within a scrollable flow view by following hyperlink-enabled method calls. Visuocode has been developed to specifically allow a programmer to create new methods juxtaposed within the flow view as, if a hyperlink-enabled, non-resolving method call is followed, an appropriate method skeleton is presented in an adjacent editor column. Due to the ability to easily create juxtaposed methods while using Visuocode, it is hypothesised that programmers should be encouraged to create better structured software during program composition tasks.

This chapter reports on a qualitative think-aloud study that investigated how programmers used the Eclipse and Visuocode environments during program composition tasks. Participants
attempted each of two tasks while using one or the other software development environment. Each task involved developing a component that implemented a provided Java abstract class. Task A involved the creation of a TelegramReporter class that counts the number of words and over-length words in a telegram; Task B involved the creation of a DateTime class that parses popular date/time strings in order to provide access to individual elements such as year, month, etc.

The format of this chapter is based on the guidelines published by Jedlitschka [JCP08]. Section 14.1 describes the technologies under investigation, alternative technologies, and related studies. Section 14.2 describes the intended experiment plan. Section 14.3 describes deviations from the experiment plan. Section 14.4 provides a summary of the participant sessions. Section 14.5 presents an analysis of the quantified navigation behaviour. Section 14.6 presents an analysis of the use of method-flow by participants. Section 14.7 discusses the results of the post-study questionnaire. Section 14.8 collects the findings of the study. Section 14.9 discusses the findings. Section 14.10 presents the conclusions.

14.1 Background

14.1.1 Technologies under investigation

The technologies under investigation are the Visuocode SDE and the Eclipse IDE. As this study focuses on program composition, this section discusses each system’s support for code creation – a more in-depth description of the two systems that focuses on software navigation is provided in Section 13.1.1.

Visuocode

Visuocode provides support for program composition by allowing the programmer to follow a non-resolving method call, which is coloured in red (instead of blue), and which creates an appropriate method skeleton in an adjacent editor column. If the new method is saved, it is added to the bottom of the appropriate source code file – this is discussed further in Section 11.1.2. Similarly, non-resolving classes are coloured in grey, which if followed present a create file dialog window that allows the class to be created in a specific project and package.

Eclipse

Eclipse supports program composition through the traditional method of just writing code in the desired editor; as well as by using its error correction capability, which highlight non-resolving methods as compilation errors, to create an appropriate method skeleton. Eclipse also provides supporting dialogs that guide a programmer through the process of creating a new class or interface.
14.1.2 Alternative technologies

Examples of other mainstream software development environments that provide explicit support for Java are IntelliJ and NetBeans. A recent research development environment that provides additional support for code creation is Code Bubbles [B+10].

14.1.3 Related studies

Related studies relevant to software navigation are discussed in Section 13.1.3. There are fewer studies of novel software development tools that involve participants modifying source code, and these often involve software maintenance tasks that require an unfamiliar codebase to be navigated, and either corrective or perfective changes to be performed. Bragdon et al. [B+10] describe a qualitative evaluation, as well as a quantitative evaluation, of Code Bubbles. The qualitative evaluation involved eliciting verbal responses from 14 participants regarding their use of the Code Bubbles tool, while the quantitative evaluation involved 20 participants who were asked to locate and fix a bug in a drawing program called ShapeDraw using either Eclipse or Code Bubbles. They used a between-subjects design, and measured the number of navigations and the number of repeated navigations per minute. They found that Code Bubbles significantly reduced the time to complete tasks, increased the number of successfully completed tasks, reduced the number of navigations per minute, reduced the time spent actively navigating, and the percentage of repeat navigations. Interestingly, they conjectured that developers used Code Bubbles “not just to avoid navigation but also to offload working memory onto concurrent views and spatial arrangements” [B+10, p. 2511].

14.2 Experiment planning

14.2.1 Goals

The goal of this study was to investigate the difference in programmer behaviour while participant’s attempted program composition tasks using a traditional development environment (Eclipse) and one that implements method-flow (Vivuscode).

14.2.2 Participants

Five participants took part in the study: four academics and one PhD level student. Each participant was considered an ‘expert programmer’ having taught one or more programming related subjects and/or being experienced with more than one programming language. Each participant had multiple years experience in either Java or a suitably similar object-oriented programming language. The academics selected were colleagues who had been known for several years including two advisors, and the student was a friend. Four of the five participants
also participated in Study 1. The participants received no compensation for taking part in the experiment.

14.2.3 Experimental equipment and materials

Each participant was seated at a desk with a 24 inch monitor, keyboard, and track pad/mouse. These were connected to an Apple MacBook laptop computer running Mac OS X 10.8. The version of Eclipse used was Eclipse Kepler, and the version of Visuocode used was 0.5.4 (configured to only have one Flow window). The version of Java SDK used was Java 7 – note that Java 8 provides additional Date/Time functionality that would make Task B trivial. Before each task, the participant was shown a short training video for the environment they were about to use that highlighted its navigation and code creation capabilities, and then was provided with a Task Sheet that described the task they were expected to attempt. After both tasks, each participant was asked to complete a short questionnaire regarding their thoughts about the Visuocode environment.

14.2.4 Tasks

Each participant was asked to complete two program composition tasks. For each task, the participant was presented with a development environment workspace that included two projects. The first project contained two task packages: one package containing an abstract class defining the required functionality, and the other containing a skeleton class that the programmer would need to complete. The second project contained JUnit tests, which the participant could run to test their implementation.

Task A involved implementing a TelegramReporter class, which accepts a stream of characters that need to be parsed into individual words and messages so that the number of words and over-length words can be counted – this task was based on a similar task described in [HS72]. Task B involved implementing a DateTime class that allows a date/time string in one of several common formats to be parsed, allowing access to individual components, i.e., year, month, day, etc. Full task sheets are available in Appendix A.

14.2.5 Hypotheses

It is believed that programmers inappropriately create overly long and complex methods because of the overhead associated with extracting code out into additional methods, as well as the difficulty of then juxtaposing their existing programming context with an editor for the new method. The research question of this study relates to whether the use of method-flow affects such behaviour during program composition tasks. The research question is:

RQ1: which environment encourages more methods to be created?
The *null hypothesis* for this research question is that participants create a similar number of methods or fewer methods while using Visuocode:

\[ H_0 = \text{METHODS( ECLIPSE )} \geq \text{METHODS( VISUOCODE )} \]

The *alternative hypothesis* is that due to the increased support for juxtaposing methods, participants create more methods while using Visuocode:

\[ H_A = \text{METHODS( ECLIPSE )} < \text{METHODS( VISUOCODE )} \]

### 14.2.6 Experiment design

The study has a within-subjects, counter-balanced experimental design – each participant used both conditions. All participants used the Eclipse environment for their first task, and the Visuocode environment for their second. This ordering was chosen due to possible learning effects noted during the previous study of software navigation (described in Chapter 13) – during that study, it appeared that participants may have altered their navigation behaviour while using Eclipse after they had used Visuocode. In Group A, two of the participants attempted Task A (Telegram Report) first and, in Group B, the remaining three participants attempted Task B (Date Time) first. This design allows the comparison of the usage of the two different environments by the same programmer. Reversing the activities for the second task makes it easier to distinguish between effects caused by the environment and effects caused by the task. Participants were assigned to groups randomly. For the analysis, the dependent variables are the number of methods created, and the number and type of navigations that participants performed while attempting the tasks. Navigation types have been previously described in-depth in Section 12.4.

<table>
<thead>
<tr>
<th>Development Environment</th>
<th>Group A</th>
<th>Group B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>Task A – Telegram Report</td>
<td>Task B – Date Time</td>
</tr>
<tr>
<td>Visuocode</td>
<td>Task B – Date Time</td>
<td>Task A – Telegram Report</td>
</tr>
</tbody>
</table>

Table 14.1: Study 2: Experiment design – a within-subjects, counter-balanced design

### 14.2.7 Procedure

The procedure included three phases: Orientation, Task Completion, and Post-study Questionnaire. During Orientation, each of the participants was seated at the computer desk and provided with an information sheet to read. It was then emphasised that the study was not intended to evaluate their programming ability, but rather to investigate how they used the environments. Participants were encouraged to treat the session as a peer programming session, and were told that they could ask any questions including questions about standard Java
classes. They were then required to sign a consent form. During Task Completion, before each task, each participant was first asked to watch a training video that described the navigation and code creation capabilities of the software development environment they were about to use, then they were asked to read a Task Sheet describing the task they would be attempting. The participants were then given 50 minutes to complete the task. The participants were given the option of a five minute break between each task. During the final phase, the participants were asked to complete a post-study questionnaire.

14.2.8 Data collection

During the completion of the tasks, each participant’s computer interaction was recorded using Apple Quicktime, which provides screen and audio capture functionality.

14.2.9 Analysis procedure

Analysis of the users sessions was performed using the approach described in Chapter 12. Additionally, it was noted at what time each participant began modifying code – signifying the end of an investigation period and the start of a modification period; and at what time the participant began debugging – identified by the time the participant indicated they were ready to run the tests because they thought their code was complete.

14.3 Execution deviations

The first execution deviation related to the amount of time that participants were given to complete each task. If the participant had not completed their first task within the allotted 50 minutes, they were told that the time for the first task was up and, if it was not apparent, asked to describe their strategy. Some participants were allowed extra time for the first task if they thought they were close to completing the task, and when completing the second task, participants were allowed to continue until they decided to stop – as such extra data could easily be truncated during analysis if required.

The second execution deviation related to attempting the tasks using Visuocode. Participants were provided with an Xcode editor window that contained the source for the ATelegramReport and ADateTime abstract classes. As the build of Visuocode that was used for the study strips out comments, this was necessary as documentation regarding the date/time string formats that needed to be parsed was contained within comments. It is thought that the only effect this would have caused is an increase in the number of implicit navigations.
### 14.4 Participant summary

The study involved five participants who are referred to as P1, P2, . . ., P5. All participants attempted their first task using Eclipse, then their second task using Visuocode. P1 and P3 carried out the tasks in the order Task A (Telegram Report), Task B (Date Time), while P3, P4, and P5 carried out the tasks in the opposite order – Task B (Date Time), Task A (Telegram Report). Appendix B includes an analysis of each participant session.

Table 14.2 shows the times at which each participant finished a particular phase of program development and moved onto the next. The (end of) investigation time corresponds to the time that the participant first modified source code by either importing a new class/package, creating a new class member, or editing a method (excepting if the participant modified the code in Visuocode solely to be able to navigate to the Javadoc for a class). The (end of) modification time corresponds to when the participant felt their solution was complete enough to justify running the tests. The (end of) debugging time, if shown, corresponds to when the participant finished debugging because either all tests passed (indicated in last column) or because they had decided to stop – if the time isn’t shown it indicates that the participant didn’t reach the debugging stage and therefore had not yet run the tests. The duration of the modification and debugging stages are indicated in brackets.

In hindsight, it is apparent that the Telegram Report task could be completed in significantly less time than the Date Time task if the participant knew about the `java.util.Scanner` class. While attempting this task, all participants reached a point where they thought their code was complete enough to start running the tests and begin debugging their code. In contrast, only two of the participants reached the debugging phase while attempting the Date Time task, and none of the participants’ solutions passed the Date Time tests. Excepting P3, during the investigation phase, every other participant took more time on the Date Time task than the

<table>
<thead>
<tr>
<th>Participant</th>
<th>Task</th>
<th>Investigation</th>
<th>Modification</th>
<th>Debugging</th>
<th>Tests Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Telegram (E)</td>
<td>04:11</td>
<td>34:04 (29:53)</td>
<td>53:17 (19:13)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>DateTime (V)</td>
<td>06:21</td>
<td>49:52 (43:31)</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>P2</td>
<td>DateTime (E)</td>
<td>07:07</td>
<td>53:56 (46:49)</td>
<td>60:43 (06:47)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Telegram (V)</td>
<td>07:04</td>
<td>19:50 (12:46)</td>
<td>21:27 (02:12)</td>
<td>Yes</td>
</tr>
<tr>
<td>P3</td>
<td>Telegram (E)</td>
<td>15:16</td>
<td>47:12 (31:56)</td>
<td>52:53 (05:41)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>DateTime (V)</td>
<td>04:08</td>
<td>40:53 (36:45)</td>
<td>81:00 (40:07)</td>
<td>No</td>
</tr>
<tr>
<td>P4</td>
<td>DateTime (E)</td>
<td>12:53</td>
<td>61:21 (48:28)</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Telegram (V)</td>
<td>02:09</td>
<td>15:50 (13:41)</td>
<td>17:50 (02:09)</td>
<td>Yes</td>
</tr>
<tr>
<td>P5</td>
<td>DateTime (E)</td>
<td>19:18</td>
<td>56:21 (37:03)</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Telegram (V)</td>
<td>06:37</td>
<td>27:47 (21:10)</td>
<td>50:18 (22:31)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 14.2: Study 2: Phase durations for each participant

Time shown in brackets is the duration of that stage.
Telegram Report task. A key problem with the Date Time task was that, even after potentially solving the problem of parsing the date/time string, each participant had to implement the accessor methods that allow the individual date and time components to be retrieved for testing. If this task is used again in the future, it could be modified so that the accessor methods are provided in the parent abstract class, or removed entirely.

Generally speaking, it can be said that all participants came very close to having a working solution for both of their tasks, and that none of the participants failed completely at their attempts. Two participants (P1 and P5) took longer than the others to complete the Telegram Report task because they did not use the java.util.Scanner class, but instead either attempted to implement their own scanner-like class or attempted to implement similar functionality within the processTelegram method. P5 stopped after recognising that they had introduced a logical flaw into their implementation that might require some time to rework in order to pass all of the tests.

All participants began the investigation phase in a similar manner. All participants first investigated the provided classes (the abstract class and the implementation class), then some also investigated the related Java classes – for the Telegram Report task, java.io.Reader and java.util.Vector; and for the Date Time task, java.lang.String and java.util.Calendar.

During the modification phase, participants would often swap between editing a method and reviewing documentation – either the abstract class that they were extending or the Javadoc for a standard Java class. For most participants, referring to the abstract class required tabbing back and forth between the two editors (← / →), however P1 and P4 both used the strategy of dragging the abstract class to the right side of the Eclipse window, which meant that it was constantly visible. Due to each opened file being presented in an individual editor tab, while using Eclipse, participants would often perform tabbing navigations to move between the editor tabs. In contrast, while using Visuocode, which does not support tabbing, and which was configured to empty the current Flow window if a direct navigation was performed, participants had to perform direct file navigations to swap between the two provided classes.

For both environments, reviewing Javadoc documentation could result in the participant swapping back and forth between the development environment and a browser window – performing lateral navigations (←→). While using Eclipse, the Javadoc for these classes was usually searched for in a browser window, however, participants P2 and P3 did occasionally perform a relative navigation into a class’s source file (as provided by Eclipse) – though often this would be followed by searching for the Javadoc in a browser. In contrast, while using Visuocode, some participants (P1, P4, and P5) made sole use of method-flow to open a class’s Javadoc documentation in a new column view, meaning they did not need to swap back and forth between different windows – P2 did initially use the method-flow for documentation, but then opened a browser window in order to be able to search the text.

During the tasks, P1, P2, and P5 were the only participants to create new software structure (methods and classes). P1 created a Scanner class for the Telegram Report task, and a parse-
method for the Date Time task. For the Date Time task, P2 created a DateTriple class, and a \textit{dateSplit} method, however these were never used. P5 created a \textit{get date parts} method for the Date Time task. P1 and P2 created their software structure using each environment’s auto-create functionality, while P5 manually created their method.

The behaviour during the debugging stage was largely determined by the environment in use. While using Eclipse, participants would run the tests, then click on the error messages to be taken to the relevant sections of code. In contrast, due to needing to use the command-line while using Visuocode, participants would swap back and forth between the Visuocode window and the command-line window.

\section*{14.5 Quantitative analysis}

The quantitative analysis consisted of counting the number of classes and methods created by each participant, as well as coding the navigations that were performed by each individual participant, and analysing whether the number of each type of navigation was dependent on either the participant, task, or environment. Additionally, repeated measures t-tests were performed to analyse the difference between the number of classes and methods created by each participant, as well as the difference between the number of implicit, direct, relative, scrolling, and editing navigations performed by each participant. Section 14.5.1 compares the number of classes and methods created by each participant. Section 14.5.2 compares the navigations performed during each participant session. Section 14.5.3 compares the navigations that were performed during each task. Section 14.5.4 compares the navigations that were performed while using each environment. Section 14.5.5 presents the results of the repeated measures t-tests. Section 14.5.6 summarises the results.

\subsection*{14.5.1 Creation of code structure}

Table 14.3 shows how many methods and classes each participant created during each task. Two participants (P3 and P4) did not create any new code structure at all – their implementations were contained purely within the stub methods that were provided to them. While P2 created the most code structure while attempting the Date Time task using Eclipse, they did not actually use the class that was created due to finding it easier to instead use the \textit{String} array returned from \textit{String.split}. While attempting the Date Time task, both P1 and P5 created methods to split either dates or times into separate digits – P1, while using Visuocode, and P2, while using Eclipse. The only participant to create a new class that was used within their solution was P1, who implemented their own \textit{Scanner} class while attempting the Telegram Report task using Eclipse.
Table 14.3: Study 2: Number of classes and methods created by each participant

TG = Telegram Report, DT = Date Time

### 14.5.2 Results by session

Table 14.4 shows the number of each type of navigation performed by each participant during each of their sessions. It is important to note that there is a dramatic variation in the duration of each session due, partially, to each participant being given more time during their second activity – i.e., until they felt they wanted to finish; but is mainly due to two participants (P2 and P4) completing the Telegram Report task quickly while using Visuocode. Due to this difference in completion times, the following sections present data in navigations per minute.

Table 14.5 shows the navigations performed by each participant divided by the duration of each session. On average, participants performed implicit navigations the most – roughly two thirds of these were lateral navigations between editors or windows. Second most-frequent were editing navigations, then direct navigations, scrolling navigations, and, lastly, relative navigations. This sort of distribution is to be expected for composition tasks as the participants

<table>
<thead>
<tr>
<th>Session</th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scolling</th>
<th>Editing</th>
<th>Total</th>
<th>Time (mins)</th>
<th>Tests Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-E-TG</td>
<td>45</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>35</td>
<td>96</td>
<td>53</td>
<td>No</td>
</tr>
<tr>
<td>P1-V-DT</td>
<td>17</td>
<td>9</td>
<td>29</td>
<td>2</td>
<td>13</td>
<td>70</td>
<td>50</td>
<td>No</td>
</tr>
<tr>
<td>P2-E-DT</td>
<td>19</td>
<td>74</td>
<td>3</td>
<td>16</td>
<td>42</td>
<td>157</td>
<td>61</td>
<td>No</td>
</tr>
<tr>
<td>P2-V-TG</td>
<td>21</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>45</td>
<td>21</td>
<td>Yes</td>
</tr>
<tr>
<td>P3-E-TG</td>
<td>36</td>
<td>26</td>
<td>1</td>
<td>40</td>
<td>20</td>
<td>123</td>
<td>53</td>
<td>Yes</td>
</tr>
<tr>
<td>P3-V-DT</td>
<td>146</td>
<td>4</td>
<td>4</td>
<td>37</td>
<td>47</td>
<td>238</td>
<td>81</td>
<td>No</td>
</tr>
<tr>
<td>P4-E-DT</td>
<td>50</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td>14</td>
<td>80</td>
<td>48</td>
<td>No</td>
</tr>
<tr>
<td>P4-V-TG</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>0</td>
<td>7</td>
<td>41</td>
<td>18</td>
<td>Yes</td>
</tr>
<tr>
<td>P5-E-DT</td>
<td>10</td>
<td>24</td>
<td>0</td>
<td>53</td>
<td>11</td>
<td>98</td>
<td>56</td>
<td>No</td>
</tr>
<tr>
<td>P5-V-TG</td>
<td>25</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>56</td>
<td>50</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 14.4: Study 2: Summary of navigations performed during each session
needed to swap between their implementation file and either documentation or the specification interface, which increases lateral navigations between a browser and code editor, direct navigations if tabbing between the two files, and increases implicit navigation if the two files are juxtaposed. The low number of relative navigations are due to the small number of files to navigate through, and the low number of scrolling navigations are due to the small size of files.

Due to the results being skewed by having a fifth participant who performed the Date Time task using Eclipse and the Telegram Report task using Visuocode, Table 14.6 shows the number of navigations performed during each treatment averaged across participants. These averages are used to produce the following two tables that analyse navigations performed by task, and then by environment.

<table>
<thead>
<tr>
<th>Treatment</th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Editing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-TG</td>
<td>1.39</td>
<td>1.66</td>
<td>0.10</td>
<td>0.53</td>
<td>1.47</td>
<td>5.16</td>
</tr>
<tr>
<td>E-DT</td>
<td>1.13</td>
<td>0.36</td>
<td>0.00</td>
<td>0.66</td>
<td>0.39</td>
<td>2.54</td>
</tr>
<tr>
<td>V-TG</td>
<td>1.83</td>
<td>1.09</td>
<td>0.82</td>
<td>0.00</td>
<td>1.05</td>
<td>4.79</td>
</tr>
<tr>
<td>V-DT</td>
<td>1.24</td>
<td>0.20</td>
<td>0.60</td>
<td>0.27</td>
<td>0.55</td>
<td>2.87</td>
</tr>
<tr>
<td>Total</td>
<td>5.59</td>
<td>3.32</td>
<td>1.53</td>
<td>1.46</td>
<td>3.46</td>
<td>15.36</td>
</tr>
</tbody>
</table>

Table 14.6: Study 2: Average of navigations performed per minute during each treatment

14.5.3 Results by task

Table 14.7 shows the summed average number of navigations performed during each task. On average, participants performed more navigations per minute while attempting the Telegram
Report task – only scrolling navigations were performed more during the *DateTime* class, likely due to scrolling through the class’s accessor methods.

Table 14.8 compares the navigations performed while attempting the Telegram Report and Date Time tasks. Excepting P1, all participants performed fewer than 10 local navigations while attempting the Telegram Report task due to the fact that, initially, the *TelegramReport* class only contained one method, and the *TelegramData* class only contained one constructor and a `toString` method. The reason that P1 was able to perform 37 local navigations was that they had added the getTelegram method to the *Telegram Report* class, and also created multiple methods within their *Scanner* class. In contrast, the participants performed more local navigations whilst attempting the Date Time task due to the numerous accessor methods in the *DateTime* class. There is no observable pattern for lateral navigations between the two tasks, which was expected as lateral navigations are mainly used if switching between the development environment and another window (such as a browser or command-line window).

All participants (except P1) performed more scrolling navigations while attempting the Date Time task due to scrolling through the *DateTime* class. P1 was able to avoid scrolling by making effective use of Visuocode Workspace Manager, and also because they did not reach the point of implementing the DateTime accessor methods. Participants attempting the Date Time task made more use of tabbing than during the Telegram Report task – the reason why P4 performed less tabbing during the Date Time task is that they used the strategy of dragging the *ADateTime* class to the right-hand side of the Eclipse window.

```
Table 14.8: Study 2: Navigations performed during each task

<table>
<thead>
<tr>
<th>Task</th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Editing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telegram Report</td>
<td>3.22</td>
<td>2.75</td>
<td>0.93</td>
<td>0.53</td>
<td>2.52</td>
<td>9.95</td>
</tr>
<tr>
<td>Date Time</td>
<td>2.37</td>
<td>0.56</td>
<td>0.60</td>
<td>0.93</td>
<td>0.94</td>
<td>5.41</td>
</tr>
<tr>
<td>Total</td>
<td>5.59</td>
<td>3.32</td>
<td>1.53</td>
<td>1.46</td>
<td>3.46</td>
<td>15.36</td>
</tr>
<tr>
<td>Mean</td>
<td>2.80</td>
<td>1.66</td>
<td>0.77</td>
<td>0.73</td>
<td>1.73</td>
<td>7.68</td>
</tr>
<tr>
<td>Std Dev.</td>
<td>0.60</td>
<td>1.55</td>
<td>0.23</td>
<td>0.28</td>
<td>1.12</td>
<td>3.21</td>
</tr>
</tbody>
</table>
```

Table 14.7: Study 2: Summed averages of navigations performed per minute during each task

```
Table 14.9 shows the average number of navigations performed while using each environment. Unexpectedly, there is very little difference between the total number of navigations per minute

## 14.5.4 Results by environment

Table 14.9 shows the average number of navigations performed while using each environment. Unexpectedly, there is very little difference between the total number of navigations per minute.
while using either environment. However, if categorised based on the kind of navigation a different story appears. Although implicit and editing navigations are similar between the two environments, more direct and scrolling navigations were performed while using Eclipse, and more relative navigations were performed while using Visuocode.

Table 14.10 compares navigations performed using Eclipse to navigations performed using Visuocode. All participants, except P3, performed more local navigations (\(\sim\)) while using Eclipse than while using Visuocode. P3 performed more local navigations while using Visuocode (36 vs 9) due to opening the \(DateTime\) class and implementing the accessor methods (instead of editing them individually). As Eclipse always opens whole source files, there is an increased likelihood for the programmer to perform a local navigation to another method. In contrast, Visuocode encourages programmers to open individual methods, which reduces the opportunity for performing local navigations.

Apart from P4, all participants performed more lateral navigations while using Visuocode than while using Eclipse. The reason that P3 has such a high number of lateral navigations is because they were the only participant to reach the debugging phase while attempting the Date Time task, and were also allowed 20 minutes longer than other participants. Both P2 and P4 used fewer lateral navigations while using Visuocode because they were both able to complete the Telegram Report task in less than 22 minutes.

Although participants performed more file navigations (\(\oplus\)) while using Visuocode than while using Eclipse, participants performed more direct navigations while using Eclipse due to participants being able to use outline (\(\odot\)) and tabbing (\(\leftrightarrow\)) navigations. While using Eclipse, participants would open a working set of files, which they would then tab between instead of performing file navigations. However, while using Eclipse, two users (P1 and P4) used a strategy of dragging an editor to the right-hand side of the Eclipse window, which allowed them to juxtapose editors, decreasing the number of tabbing navigations necessary,
but increasing the number of lateral navigations. All participants used tabbing navigations (← / →) while using Eclipse – such navigations are not available in Visuocode. Excepting P1, while using Eclipse, all participants performed more tabbing navigations than file, or outline, navigations.

Relative navigations (⇒ / → / ←) were used by all participants while using Visuocode to either view Javadoc documentation or create new code structure. In contrast, only P2 and P3 used relative navigation while using Eclipse – P2 extensively, while P3 only once to look at the Reader class.

Surprisingly, only one participant used scrolling navigations extensively while using Visuocode – P3 scrolled through the DateTime class. The other participants would instead navigate long classes by selecting individual methods from the class outline in the Workspace Manager. In contrast, scrolling navigations were performed by all of the participants while using Eclipse.

It appears that participants tended to initiate more edits while using Eclipse than while using Visuocode. While using the Visuocode environment, all participants initiated a number of edits in the range 7 – 13 except for P3 who made 47 edits due to first implementing the accessor methods of the DateTime class. The same is true for the total number of navigations.

In total, participants performed 104 more navigations while using Eclipse than while using Visuocode (554 vs 450),

### 14.5.5 Results of the repeated measures t-tests

A repeated measures t-test can be used to investigate the average difference in the number of classes and methods created, as well as the number of each kind of navigation performed by each individual participant while using each development environment. Two tests were performed to identify whether the use of a particular environment is associated with a difference in the number of classes or methods created. Five tests were performed to identify whether the use

<table>
<thead>
<tr>
<th>Structure type</th>
<th>Mean</th>
<th>Std Dev</th>
<th>t-Test score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>-0.400</td>
<td>0.548</td>
<td>-1.633</td>
<td>0.1778</td>
</tr>
<tr>
<td>Methods</td>
<td>-2.000</td>
<td>2.550</td>
<td>-1.754</td>
<td>0.1542</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Navigation kind</th>
<th>Mean</th>
<th>Std Dev</th>
<th>t-Test score</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit</td>
<td>12.200</td>
<td>58.764</td>
<td>0.464</td>
<td>0.6666</td>
</tr>
<tr>
<td>Direct</td>
<td>-19.600</td>
<td>26.913</td>
<td>-1.628</td>
<td>0.1787</td>
</tr>
<tr>
<td>Relative</td>
<td>8.600</td>
<td>12.621</td>
<td>1.524</td>
<td>0.2022</td>
</tr>
<tr>
<td>Scrolling</td>
<td>-16.000</td>
<td>21.656</td>
<td>-1.652</td>
<td>0.1738</td>
</tr>
<tr>
<td>Editing</td>
<td>-6.000</td>
<td>22.282</td>
<td>0.602</td>
<td>0.5795</td>
</tr>
</tbody>
</table>

Table 14.11: Study 2: Summary of results from repeated measures t-tests

Visuocode versus Eclipse
of a particular environment is associated with a difference in implicit, direct, relative, scrolling, and editing navigations. The hypotheses for the tests are all the same except for the type of navigation under analysis. The null hypothesis \((H_0)\) is that the mean of the Eclipse and Visuocode navigation differences is equal to zero, while the alternative hypothesis \((H_a)\) is that the mean is not equal to zero.

Since all tests are repeated measures tests with \(n = 5\), a two-tailed t-distribution is used with degrees-of-freedom = 4. Using an \(\alpha = 0.05\), the null hypothesis is rejected if the test statistic is less than -2.776 or greater than 2.776.

Table 14.11 shows for each structure type and navigation kind, the results of a two-tailed repeated measures t-test that analyses the difference in navigations performed by each participant. The results show that for each structure type, the differences in structures created by each participant are no more than what can be explained by chance alone; and that for each navigation type, the differences in the number of navigations performed by each participant are no more than what can be explained by chance alone.

### 14.5.6 Summary of results

Repeated measures t-tests that compared the number of classes and methods created during participants sessions revealed that there is no difference between the number of each type of structure created than what can be explained by chance alone. The analysis of the navigations performed during each task shows that participants performed more navigations per minute during the Telegram Report task than during the Date Time task for every kind of navigation except scrolling navigations. The analysis of the navigations performed while using each environment shows that participants performed more relative navigations while using Visuocode; and more direct and scrolling navigations while using Eclipse. However, repeated measures t-tests that compared the frequencies of direct, relative, and scrolling navigations revealed that there is no difference between the number of each type of navigation performed by each participant other than what can be explained by chance alone.

### 14.6 Qualitative analysis of Visuocode (only)

Every participant except P3 spent a short amount of time performing direct navigations using the Workspace Manager in order to orient themselves to the provided source code. Every participant except P3 then used method-flow to open the Javadoc documentation for a relevant class in the flow view, and all of these participants continued to use method-flow in this way except P2 who opened a web browser so that they could use text search. While using Visuocode, P1 was the only participant to seriously use method-flow for developing new code as the other participants attempted to implement their solutions using the existing stub method.
14.7 Post-study questionnaire

Table 14.12 shows a summary of the responses to the post-study questionnaire. The post-study questionnaire asked participants which environment they preferred and why, and also asked them to suggest five features they would want implemented in Visuocode before evaluating it again. When asked their preference of environment, most participants either preferred Eclipse (due to its comprehensive functionality), or were unsure. Only one participant preferred Visuocode – they wrote “I preferred Visuocode vs Eclipse because the Eclipse interface is very noisy (lots of animation while you edit is very distracting)”. The most requested feature was method auto-suggest (3 respondents), where the environment suggests what methods are available to be called on a class. The next most requested feature was additional error localisation and support (2 respondents) – similar to the annotations that Eclipse provides in the margin of the code editor. Other features requested included parameter names (in addition to types) in the method signatures of the Workspace Manager, incremental compilation and testing support, search support, hyper-link enabled system classes, and auto-indent. P3 also requested that it be easier to navigate between methods using the cursor, and for there to be additional keyboard shortcuts. In contrast to the software navigation post-study questionnaire described in Chapter 13, this questionnaire asked the participants whether they would consider evaluating the environment again in the future, rather than whether they would use it in the future. With this change in wording, respondents were generally positive about whether they would re-evaluate it in the future, with three responding yes, and the other two responding they were unsure.
<table>
<thead>
<tr>
<th>Question</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which environment did you prefer?</td>
<td>Neither</td>
<td>Eclipse</td>
<td>Visuocode</td>
<td>Neither</td>
<td>Eclipse</td>
</tr>
<tr>
<td>Requested features</td>
<td>Method Auto-suggest</td>
<td>Eclipse Plug-in</td>
<td>Inter-method Navigation</td>
<td>Error Support</td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td>Method Auto-suggest</td>
<td>Method Auto-suggest</td>
<td>Method Auto-suggest</td>
<td>Parameter Names</td>
<td>Freeform text</td>
</tr>
<tr>
<td>If requested features were implemented, would you evaluate in the future?</td>
<td>Yes</td>
<td>Maybe</td>
<td>Unsure</td>
<td>Yes</td>
<td>Unsure</td>
</tr>
</tbody>
</table>

Table 14.12: Study 2: Results of the post-study questionnaire
14.8 Findings

The research question was: “which environment encourages more methods to be created?” The null hypothesis ($H_0$) that participants would create a similar number of methods or fewer while using Visuocode than while using Eclipse, has been found to hold. In total, participants created 12 new methods while using Eclipse, but only two new methods while using Visuocode. Analysis of the number and types of navigations performed by participants reveals that the navigation behaviour of participants was similar to that found in the first study of software navigation – participants performed more relative navigations while using Visuocode than while using Eclipse, and participants performed more scrolling navigations while using Eclipse than while using Visuocode – however it should be noted that the difference was not statistically significant when measured using repeated measures t-tests for each navigation type for each participant. In summary, the findings of this study are:

1. Participants created more software structure while using Eclipse.
2. Participants performed more relative navigations while using Visuocode.
3. Participants performed more direct navigations while using Eclipse.
4. Participants performed more scrolling navigations while using Eclipse.
5. Participants’ navigation behaviour was affected by the task.

14.8.1 More software structure created while using Eclipse

Referring to Table 14.3 we see that while using Visuocode, only one participant created two extra methods. In contrast, while using Eclipse three of the participants created 12 new methods, some of which were within 2 new classes. However, from analysis of the participant sessions, it appears that the decision to create or not create new methods was influenced more by the personal preference of the individual participant, and the task being attempted, than whether they were using Visuocode. Additionally, repeated measures two tailed t-tests showed there was no difference in structure created other than what can be explained by chance alone.

14.8.2 More relative navigations while using Visuocode

Participants performed more relative navigations while using Visuocode than while using Eclipse. In total, participants performed 50 relative navigations while using Visuocode, compared to 7 while using Eclipse – 1.43 vs 0.10 per minute. While using Visuocode, the number of relative navigations (50) also exceeded the number of scrolling navigations (39) – 1.43 vs 0.27 per minute.
14.8.3 More direct navigations while using Eclipse

Participants performed more direct navigations while using Eclipse than while using Visuocode. In total, participants performed 146 direct navigations while using Eclipse, but only 48 while using Visuocode – 2.02 vs 1.30 per minute. The reason for this difference seems to be due to Eclipse users opening a working set of source code files that they then tab between and scroll through. While participants did perform more file navigations while using Visuocode (48) than while using Eclipse (23), these were outnumbered greatly by tabbing navigations.

14.8.4 More scrolling navigations while using Eclipse

Participants performed many more scrolling navigations while using Eclipse than while using Visuocode. In total, participants performed 119 scrolling navigations while using Eclipse, but only 39 while using Visuocode – 1.19 vs 0.27 per minute. In fact, only one participant used SCROLL UP or DOWN navigations while using Visuocode. As Visuocode, like Eclipse, is able to open all methods of a class within an editor column if the programmer choses, it cannot be concluded that this difference is due to Visuocode not requiring scrolling. This distinct difference is thought to be due to a difference in behaviour between the Visuocode Workspace Manager and the Eclipse Package Explorer. In Visuocode, classes are represented directly below packages, and if a class is expanded it immediately shows all of the class’s members. In Eclipse, classes are represented within a containing source code file, which means that to access the members of a class, first the source code file must be expanded to reveal classes, then a class must be expanded. While using Eclipse, participants would often instead double click the class causing the file to open – possibly in order to activate the outline view. Additionally, if a method is selected in Visuocode, only that method appears in the Flow window whereas Eclipse shows the entire source file, which encourages scrolling behaviour.

14.8.5 Navigation behaviour was affected by task

It appears that navigation was also heavily influenced by the task and codebase being navigated. In particular, the fact that the DateTime class has numerous accessor methods resulted in numerous local and scrolling navigations, especially in Eclipse, which encourages participants to open a source file and scroll down through it.

14.9 Discussion

This study investigated whether providing additional support for software structure creation would encourage programmers to create better structured software. Due to the suspicion that poor software structure is often caused by programmers creating methods that are overly large
and complex, the number of methods created was used as a proxy for measuring the quality of software structure.

It was expected that participants would create more methods while using Visuocode due to its explicit support for juxtaposing methods within a flow view. The opposite turned out to be the case. Participants created more methods while using Eclipse than while using Visuocode (12 vs 2). However, analysis of the participant sessions suggests that this result may be due to a range of factors beyond the environment. Firstly, the tasks may have confounded results. If the participant was aware of the \texttt{java.util.Scanner} class, they were able to complete the Telegram Report task with less code than otherwise – three of the participants attempted this task using Visuocode, and two of them used the \texttt{Scanner} class. While attempting this task, P1 was the only participant to create additional methods, and they did so due to implementing their own \texttt{Scanner} class while using Eclipse. If they had attempted the task using Visuocode, it is possible that they would have created a similar number of methods. Additionally, while attempting the Date Time task, P2 created two methods (and a new class) which were not used due to \texttt{String.split} returning an appropriate data-structure. Secondly, the tasks set may have been too small for a valid result. New methods were created in only four of the 10 participant sessions by three of the five participants. While using Eclipse, new methods were created in only three of the five sessions. This indicates that the tasks were small enough that some participants did not feel the need to create new methods regardless of the environment. Thirdly, participants may have created more methods while using Eclipse because they were more comfortable with that environment. It is possible that, as it is a prototype environment, the participants may have not wanted to push Visuocode. Lastly, the number of participants is thought to have been too few to yield consistent results.

While using Visuocode, participants extensively used method-flow to view Javadoc documentation associated with the Java library classes they were using. In contrast, while using Eclipse, participants would often open Javadoc in a separate web browser window, and either swap back and forth between the windows or rearrange the windows to not overlap so they did not need to view documentation within an Eclipse tab. Thus, method-flow allowed participants to view their programming context and the documentation side-by-side without any window management burden. In particular, a common request from participants was to allow the navigation of Javadoc documentation in the same manner as between methods – currently, once a web browser is opened in the flow view, following page links causes that web browser to be updated with the new page.

While using Visuocode, an Xcode window was used to display either the \texttt{ATelegramReport} or \texttt{ADateTime} class source code file as Visuocode currently strips out Javadoc style comments from source code. This does not appear to have affected the available screen area very much as the participant usually had the window sized fairly small (1/3 width of screen) and positioned in the corner of the screen in such a way that it was partially visible. Navigation-wise, swapping back and forth between the Xcode window and Visuocode would have increased the number of lateral
navigations which were not relevant to the findings. Also, apart from P3 who implemented all of the accessor methods of \textit{ADateTime} first, the other participants rarely referenced the Xcode window. It might be argued that participants did not have the benefit of the Xcode window while using Eclipse. P1 and P4 used the strategy of dragging the abstract classes to the left side of the screen so, for them, it would have increased their implicit navigations in a similar way to Visuocode. However, P2, P3, and P5 had the abstract file open in a tab and did swap back and forwards between it and their implementation file, which would have increased the amount of tabbing that they performed, and therefore the number of direct navigations.

Interestingly, the findings of this study have repeated those of the previous study – navigation behaviour was influenced by the task being attempted, participants performed more relative navigations while using Visuocode than while using Eclipse, participants performed fewer direct navigations while using Visuocode than while using Eclipse, and participants performed many fewer scrolling navigations while using Visuocode than while using Eclipse. However, there remains the possibility that the variation in session lengths has affected the characterisation of software navigation in a way that is not accounted for by averaging the navigations. In particular, those with the shortest sessions lengths usually also performed debugging, while those with the longest usually were performing development right up to the end of the session.

In hindsight, it is believed that the tasks set were too small to require the participants to create enough software structure to test the hypothesis. However, the time required to complete the tasks was close to that planned (apart from the unexpected use of the \textit{Scanner} class). This suggests that in order to test the hypothesis, a study with a much longer session length would be required, which would be difficult for this format of study.

\subsection*{14.9.1 Threats to validity}

Threats to validity may include threats to: construct validity, which relate to whether the measures taken during an experiment are as meaningful as supposed in the real world; internal validity, which relate to the extent the treatments were responsible for the effects observed; external validity, which relate to how findings of the study may be generalised to other participant populations; and conclusion validity, which relate to whether statistical tests, upon which conclusions have been based, have been carried out correctly \cite{JCP08}.

\subsubsection*{Construct validity}

For this study, the first threat to construct validity relates to what degree the number of methods created can be used as a proxy measure for the quality of software structure. It is important to note that this is only claimed as a reasonable measurement for small programs that are being created from scratch. The second threat to construct validity relates to the meaningfulness of quantifying the number of each type of navigation performed by a participant.
Internal validity

For this study, the threat to internal validity relates to the extent that the development environments used were responsible for the different software structure created, and the difference in navigation behaviour observed. The main threat to internal validity is the effect of individual participants, as well as the tasks, on the results. Another potential threat to internal validity is the novelty of the conditions involved. As previously mentioned, it is possible that participants may avoid stressing the prototype environment and therefore create less code structure.

External validity

For this study, the two main threats to external validity are the artificialness of the tasks attempted and the motivation of participants. Pragmatically, both tasks needed to be small enough to be completed in an hour, and simple enough that they would not need extensive domain knowledge of any sort. This limits the real world applicability of results, and also may affect the rigour with which the participants normally develop software.

Conclusion validity

For this study, the conclusion validity relates to the confidence that the repeated measures t-tests were carried out correctly and are applicable for analysis of a problem of this sort. In this instance, the statistical tests showed that the variation between each participant’s navigations using each environment was no more than what can be explained by chance alone. However, it is stressed that the statistics performed for this study have been used to support the qualitative analysis, which mainly contributed toward the conclusions.

14.9.2 Lessons learned

The results of this study highlight the importance of taking great care when developing tasks. In hindsight, even though the size of the tasks successfully matched the desired duration for programming sessions, the amount of code produced was too small to satisfactorily test the hypothesis of the study - that Visuocode supports software structure creation. However, while some participants were able to complete the Telegram Report task and successfully run the tests quite quickly, none of the participants were able to successfully run the tests for the Date Time task (though some were very close). It appears that, in order to test the hypothesis of this study, larger and longer running tasks will be needed.

14.10 Conclusions

It is concluded that during program composition, the method-flow visualisation aids programmers by allowing them to display documentation within an adjacent web browser column within
the flow view. Even though the results revealed that participants created fewer methods while using Visuocode than while using Eclipse (not more), it is conjectured that the tasks were not big enough to encourage or require the participants to create code structure because no new methods were created during six of the ten participant sessions, and the repeat measures t-test indicated that the variance was not greater than what can be explained by chance alone. Therefore, it is currently not possible to draw any conclusions regarding whether Visuocode encourages the development of higher quality software structure or not. Due to the extensive use of method-flow by the participants, it is concluded that method-flow, and the explicit support for software navigation it provides, may improve programmer performance if adopted by other development environments, but whether it does encourage the creation of better quality software structure is not yet apparent. Finally, it is notable that two of the three participants who successfully completed the Telegram Report task did so using the Visuocode environment.
Part V

Conclusion

This part concludes the thesis by providing a discussion that draws together the key points of the thesis then presents the main conclusions.
Chapter 15

Discussion

It has been recognised that while there has been extensive research into the cognitive processes used by programmers to form their mental model of the function of software, there has been little research into how the structure of software – referred to as codespace – is represented within spatial memory as a cognitive map [CFO05]. This may have contributed toward software development environments not providing enough support for the navigation of software structure, which results in programmers becoming disoriented and losing task awareness [dAM06]. Recent studies have found that programmers often use a two-phase strategy that includes an opportunistic exploration phase and a systematic traversal phase [KAM05, K+11, KKK+13]. During the exploration phase, programmers perform direct navigations into files to find potentially relevant sections of code – referred to as anchor points; then during the traversal phase, programmers navigate the local call graph from each anchor point to form a working set of source code files. Similar behaviour was observed during the two formal studies carried out as part of this project. It is during the traversal of the call graph that programmers often become disoriented. A key problem is that the existing programming context is usually replaced by each method navigated to during navigation – either by the editor scrolling, or by a new file being opened in place of the current editor (even if in a new tab). During the formal studies, it was observed that while using Eclipse, participants often tried to remain oriented by using the editor tabs as breadcrumbs, however, at least one participant lost track of which tab belonged to which file, causing them to have to reorient themselves using the ‘back’ button.

15.1 Related research

Hoping to leverage spatial memory, various software understanding tools have been developed that attempt to provide a spatial representation of software by representing source code files as boxes that are displayed in a spatially consistent manner that may then be opened to view their source code. However, the spatial representation used by these tools is not based on the emergent structure of the software – it is either based on the file structure of the software [MK88], or requires manual annotation of a suggested structure [SBM+02]. It is believed that
a key problem with such tools is that, similar to mainstream development environments, when
the user opens/views a source file, the spatial representation disappears and is replaced with
the source code. Such behaviour is likely to interfere with the integration of each source file
into a comprehensive cognitive map within long-term spatial memory as all source files appear
in a common spatial location. To mitigate this problem, some tools allow the user to zoom
in and out of source files to provide a more seamless spatial experience [DR10]. Using the
terminology discussed in Chapter 2, these tools can be recognised as providing an allocentric
view of software.

A challenge is that for any non-trivial software project the amount of source code is always
more than can fit on a single computer screen. Recent prototype environments have sought to
mitigate this problem in various ways. Code Bubbles refers to itself as a working set environ-
ment, and it allows methods (as bubbles) to be arranged in groups on a horizontally scrollable
workspace – Code Bubbles also provides additional support for relative navigation including
allowing called methods to be ‘budded off’ an existing method [BRZ+10a, B+10]. Code Canvas
is a similar system except that source code fragments can be arranged on a number of separate
surfaces [DR10]. The Patchworks system presents a $3 \times 2$ grid of editors that exist within
a never-ending ribbon of editors – the programmer is able to zoom out to see all previously
arranged editors on the ribbon, then can zoom back in to another chosen set [HF14]. These
systems seek to solve the problem of what source code to show by having the user specify a
subset of the codebase. These systems are recognised as providing explicit support for implicit
local navigations between visible methods, but (excepting Code Bubbles) provide little extra
support for relative navigation. In contrast, other systems augment a traditional code editor
with views that allow the programmer to more easily navigate between methods adjacent on
the call graph (Stacksploiter [K+11]), or represent the current code branch using a combination
lock metaphor (Blaze [KKK+13]) – however, while these systems provide explicit support for
relative navigation, it is thought that they do not fully support the formation of a cognitive
map because each method navigated to obscures the last.

15.2 Theory

This project has taken a different approach that is based on trying to provide codespace with a
canonical dimensional spatial structure that it normally lacks; and then implementing a visu-
alisation technique that uses this structure to provide a more consistent spatial representation
of methods. The key principle is that the emergent structure of software is rooted in the main
entry point of a program. From the main method, the composition hierarchy and call graph
represent a canonical structure of software. If a programmer explores down a branch of the call
graph, the methods traversed remain the same unless the structure of that branch is modified.
The intent is to provide better support for the integration of journeys along paths through the
emergent structure of software into a cognitive map within long-term spatial memory. A further
dimension is introduced through polymorphism due to class inheritance, and the implementation of interfaces. The concept of Software Dimensions allows necessary methods such as parent constructors to also be displayed within the flow view so that they too can be incorporated into the spatial cognitive map.

This project has introduced the concept of a visuo-spatial programming interface in order to identify issues related to leveraging spatial memory in software development environments. It is important that such interfaces maintain the spatial consistency of programming artefacts — such as methods or classes — within the environment so that they can be more easily integrated within a cognitive map in long-term spatial memory. The method-flow visualisation technique has been developed, which allows editor columns to be juxtaposed within a scrollable flow view in order to ensure that the methods that a programmer traverses, and therefore the branch that is traversed, are integrated into a cognitive map of the software structure. Method-flow increases the visual momentum of environment, allowing a programmer to explore down a call graph branch leaving their existing programming context unaffected — at any time they are able to scroll back to refresh their visuo-spatial memory regarding the position and content of any traversed editor columns. The concept of Software Dimensions has also been proposed to guide how software may be represented within such an environment, in order to mitigate the hyperspace problem. During this project, method-flow was implemented within two software development tools – Code-flow and Visuocode.

15.3 Implementations

The initial implementation – Code-flow – is a software exploration tool for software written in the Java programming language. The benefit of creating a software exploration tool was that the implementation could be simpler, and it was not necessary to worry about the myriad issues that need to be considered for a software development environment. As well as demonstrating the method-flow concept, the implementation highlighted aspects of the Java programming language that were ignored in the second implementation for simplicities sake, such as class-style enumerations. Also, due to experience with Code-flow, the second implementation was designed to have a separate Workspace Manager window that could then create any number of flow windows, however, this capability was not used during the evaluations. In hindsight, it would have been advantageous to perform an earlier empirical evaluation of Code-flow, instead of waiting for Visuocode to be completed, so that the method-flow visualisation technique could have received some early formal validation, and to gain experience in performing qualitative studies.

The second implementation – Visuocode – is a prototype software development environment also for software written in the Java programming language, but was itself implemented in Objective-C++. In contrast to mainstream IDEs such as Eclipse, Visuocode does not support features such as incremental compilation, error warnings, smart indenting, or refactoring.
However, due to the necessity of resolving and presenting hyperlink-enabled method calls, Visuocode does provide a form of syntax highlighting that indicates syntactic errors if keywords are misspelled, and also when misspelled class and method names do not resolve.

15.4 Formal evaluation

During the software navigation study, it was observed that participants first built up their understanding of the software’s composition hierarchy, then investigated the flow of execution through it. While using Eclipse, participants would use the Package Explorer to locate possibly relevant classes – each of which they would open in a tabbed editor, then scroll down through to investigate its methods. After forming a working set of relevant files, they would then perform mental execution to better understand how the different classes interacted. In contrast, while using Visuocode, participants would often expand a class within the Workspace Manager to view what methods were contained within the class before deciding whether or not to open it, whereas, while using Eclipse, participants would only rarely expand the class in the Package Explorer but might later refer to its auxiliary outline view. Interestingly, while using Eclipse, it seemed that participants would often double-click a class node within the Eclipse Package Explorer to open the class in an editor because Eclipse requires the user to expand both the source code file to reveal the class, then expand the class to view its outline. This would often result in the participant opening the editor, then scrolling down through the file. Due to this, during both formal studies, participants performed many more scrolling navigations while using Eclipse than while using Visuocode. In contrast, while using Visuocode, programmers were more likely to open the class outline in the Workspace Manager and then navigate directly to a single method, which prevents scrolling to other methods.

During the software navigation tasks, participants appeared to build their mental models in a different way depending on the environment used. As previously described, while using Eclipse, participants would open possibly relevant files using the Package Explorer, then scroll down through them in order to better understand how they related to the rest of the software codebase. Often they would find a reference to another class, which they would then open and digest in a similar way. This style of navigation usually reflects a traversal of the composition hierarchy, and results in the programmer’s mental model being structured in a similar way. In contrast, while using Visuocode, participants were far more likely to use the method-flow to build up their understanding of the software based on the flow of execution along the call graph. However, it appears that if a programmer’s mental model is built up using the call graph, they may later feel misled if they come across methods that they were previously unaware of within a class. During the self-evaluation, this issue was identified as feeling ‘out-of-control’. During the formal evaluations, several participants became confused by only seeing one method in an editor column. One participant, scrolling through source code in Eclipse, expressed surprise at finding methods that they had missed previously while using Visuocode. Another participant
remarked at the end of their session that “it took a while to get used to not looking at whole source files”. During the post-study questionnaire, one participant wrote that they felt misled by only seeing single methods, and suggested showing the other methods in elided form. The term ‘blinckered’ is used to describe this effect, as like a blinkered horse that can only see the road in front, the participants could only see the single method and not any methods nearby in source code files.

During the program composition tasks, the participants navigated between the reference abstract classes that needed to be implemented, the Javadoc documentation associated with system classes related to the implementation, and the JUnit test classes; as well as their implementation classes. In particular, this study highlighted how the task can affect navigation behaviour as the Date Time task was associated with increased local and scrolling navigations due to the numerous accessor methods that needed to be created within the DateTime class. While using Eclipse, two participants used the strategy of dragging the interface source files to the right-hand side of the Eclipse window, which resulted in them performing many fewer tabbing navigations than the other participants but more lateral navigations. While using Visuocode, even though method-flow was only used by one participant to create new software structure, it was used extensively to view Javadoc documentation alongside the current programming context. Interestingly, during the program composition study, three participants performed no scrolling navigations while using Visuocode. This was due to them instead using the class outline provided by the Workspace Manager to open individual methods.

It was previously thought that, while using Eclipse, participants would favour direct file navigations (using the Package Explorer) and that participants would favour relative navigation while using Visuocode. It appears that such a rigid dichotomy is not correct as it does not properly take into account tabbing navigations – some participants performed a similar number of tabbing navigations to file navigations while using Eclipse. What is apparent is that while using Eclipse, programmers initially perform direct navigations to form a working set of editor tabs that they then continue to swap back and forth amongst. However, the results also show that this usage of editor tabs encourages a programmer to perform far more navigations than if such tabbing capability is not present.

15.5 Good versus bad navigations

Up until this point there has been no discussion about whether any specific forms of navigation are ‘good’ or ‘bad’, however, the question must be raised as to which navigation actions support the programmer with their task – ‘good’ navigations – and which navigation actions might be considered a hinderance – ‘bad’ navigations. An assumption is that the fewer navigations performed during a programming task the better because, presumably, if a navigation did not have the desired effect the programmer would need to perform more navigations in order to reach the desired location. Also, if the programmer spends less time navigating they should
suffer fewer distractions while navigating and have more time available for comprehending or composing code. While more research is needed that evaluates the effectiveness of each type of navigation, observations suggest the following:

1. Implicit local navigation between methods and lateral navigations between method editors are more effective than lateral navigations between different applications, e.g., between a development environment and a web browser.

2. If opportunistically searching for classes, direct navigations are more appropriate than relative navigations.

3. If traversing the call graph, relative navigations are more effective than direct navigations if the environment provides explicit support to prevent disorientation, such as with method-flow.

4. Scrolling navigations can lead to serendipitous discovery of relevant sections of code, but at the cost of a lot of time spent aimlessly scrolling through classes.

5. Although useful when swapping between a small number of source code files, a lot of time can be wasted tabbing between editors searching for a desired editor.

Interestingly, participants would often navigate pseudo-spatially within the environment, i.e., they would click on the previous tab without checking the name of the file, or would click on an item in the file browser without checking its name, which would cause confusion if they were presented with a source code file they did not expect. Observations from the participant sessions indicate that a lot of time is wasted scrolling through classes while trying to understand the relationships between classes.

15.6 Reflection on empirical approach

Any empirical approach has tradeoffs that need to be considered carefully – these are discussed further in Chapter 12. The following benefits and limitations have been identified regarding the empirical approach taken for this project.

15.6.1 Benefits

The benefits of the empirical approach relate to the rich data obtained from carrying out qualitative studies, and the way that the quantitative characterisation is able to complement the qualitative analysis.

The most important benefit of the approach is that the think-aloud participant sessions provided rich data that could be thoroughly analysed. Controlled experiments that seek to obtain statistical significance between quantitatively measured metrics such as the number of
answers correctly answered, or the time taken to complete a task, do not provide any insight into how participants are actually using the tools being measured, or even if the participants are using the features that are meant to make those tools novel. In particular, the large difference in programmer performance on different tasks often confounds such analysis.

During Study 1, the non-prompted think-aloud protocol worked very well. The mere presence of the experimenter in the room encouraged the participants to talk naturally during their sessions due to it being somewhat uncomfortable being silently watched as one completes a task. During Study 2, the peer programming style communication between participant and experimenter also worked well.

The counting of the navigations of each type performed by each participant provided a useful characterisation that, together with the qualitative analysis provides insight into how the different tasks and environments affect the navigation behaviour of participants.

Finally, the qualitative analysis provides a rich insight into software navigation and creation behaviour exhibited by participants. For example, during the software navigation study, one participant carefully pruned unneeded editor tabs to prevent becoming confused by extraneous tabs; and during the program composition study, two participants used the strategy of dragging the abstract class files to the right-side of the Eclipse window so that they would always be visible (refer to Appendix B and C).

15.6.2 Limitations

The limitations of the approached relate to the external validity of the tasks, and the lack of an established theoretical basis for claiming that a change in any type of navigation type also corresponded to an increase (or decrease) in programmer productivity.

The task size for Study 1 is considered small by many standards. TextEditor++ contains 1796 LOC, while JavaChat contains 2768 LOC. In practice, this also means that participants do not need to navigate very far through the code in order to complete the tasks. However, considering that most software systems are designed in a modular fashion, and that an average maintenance task might involve making changes contained within a single module, the task size may not be as unrealistic as supposed.

In hindsight, the tasks set for Study 2 were not large enough for there to be a statistically significant difference between the number of classes and methods created by participants. As the duration of each participant’s sessions were approximately what were planned – except for those who completed the Telegram Report task quickly due to knowledge of the Scanner class – re-running the study with larger tasks would also require finding participants willing to participate in much longer sessions.

A common issue with any controlled experiment (as opposed to a field study) is the external validity of setting a task that the participant has little motivation for doing well, or in a manner that reflects the need to return to the task at a later date. In particular, there is the risk that
the participants ‘try out’, or persist with the use of, environment features that they would not normally use, or would give up using under normal circumstances – this also applies to the control environment.

Regarding the categorisation of navigations into types, in hindsight, tabbing navigations should be categorised into their own set of ‘working set’ navigations that might also include navigations related to bookmarks and other similar ‘recommender’ functionality. Also, lateral navigations between different applications should be categorised differently to lateral and local navigations within the same environment.

Finally, as alluded to in Section 15.5 there is currently no theoretical basis for believing that increased relative navigations are associated with increased productivity. The biggest claim that can be made is the an increase in relative navigations reflects programmers being supported in navigating the way they want to navigate but are prevented from when method-flow functionality is unavailable.

15.7 Limitations of Visuocode

Due to being a standalone prototype development environment that was developed entirely by the author during the course of this project, Visuocode does have some limitations that may have affected the results of the empirical studies.

15.7.1 Visuocode does not support the 3rd Software Dimension

Due to the desire to empirically evaluate the visuo-spatial aspects of method-flow as soon as possible, Visuocode was implemented to only support the first and second Software Dimensions (described in Chapter 9). The first Software Dimension reflects the representation of a method’s source code, while the second Software Dimension reflects composition and function encapsulation by representing called methods in an adjacent editor column. The third dimension, which is not currently supported, represents polymorphic relationships such as inheritance from superclasses, as well as implementation of interfaces. It is now felt that the best way to implement such relationships is to include the attributes of ancestor classes within the lists for the current class. For example, the list of members would have a section for each ancestor class, then a section containing the members of the current class. Similarly, the methods area would contain all methods belonging to each class in different sections. If a method call was followed such that only one method would normally be shown, if the method is a constructor, any ancestor constructors would be shown above it (as they would be executed first), and if the method call first calls an overridden method, that method would be similarly shown above, otherwise, it would be shown in an adjacent editor column.
15.7.2 Source code lag

While using Visuocode, navigation behaviour may have been affected by the amount of time that it took to display methods if a whole class was displayed. During the software understanding study, there was considerable lag as Visuocode needed to resolve all method calls within all method editors if opening a whole class for the first time – the content was cached for each method after being displayed the first time. This may have encouraged the behaviour exhibited of reviewing the class outline in the Workspace Manager instead of opening a class and scrolling through it. Table shows for each participant each occasion that a participant experienced lag due to Visuocode attempting to show an entire class – note that P2 used a navigation style that did not produce lag as they avoided opening entire classes.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Class</th>
<th>Timestamps</th>
<th>Lag (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>MainFrame</td>
<td>14:11 - 14:33</td>
<td>22</td>
</tr>
<tr>
<td>P1</td>
<td>ClientScreen</td>
<td>01:29 - 01:48</td>
<td>19</td>
</tr>
<tr>
<td>P1</td>
<td>TextEditorPane</td>
<td>13:03 - 13:12</td>
<td>09</td>
</tr>
<tr>
<td>P3</td>
<td>TextEditorPane</td>
<td>10:57 - 11:10</td>
<td>13</td>
</tr>
<tr>
<td>P3</td>
<td>ClientScreen</td>
<td>00:23 - 00:39</td>
<td>16</td>
</tr>
<tr>
<td>P4</td>
<td>Page</td>
<td>01:10 - 01:15</td>
<td>05</td>
</tr>
<tr>
<td>P4</td>
<td>ClientScreen</td>
<td>38:42 - 38:56</td>
<td>14</td>
</tr>
<tr>
<td>P4</td>
<td>ClientManager</td>
<td>41:42 - 41:45</td>
<td>03</td>
</tr>
<tr>
<td>P5</td>
<td>MainFrame</td>
<td>00:34 - 00:57</td>
<td>23</td>
</tr>
<tr>
<td>P5</td>
<td>MainFrame</td>
<td>02:29 - 02:39</td>
<td>10</td>
</tr>
<tr>
<td>P5</td>
<td>TextEditorPane</td>
<td>09:22 - 09:31</td>
<td>09</td>
</tr>
<tr>
<td>P5</td>
<td>ClientScreen</td>
<td>15:24 - 15:38</td>
<td>14</td>
</tr>
<tr>
<td>P5</td>
<td>ClientScreen</td>
<td>15:41 - 15:48</td>
<td>07</td>
</tr>
<tr>
<td>P5</td>
<td>ClientScreen</td>
<td>16:33 - 16:39</td>
<td>06</td>
</tr>
<tr>
<td>P5</td>
<td>ClientScreen</td>
<td>17:32 - 17:38</td>
<td>06</td>
</tr>
<tr>
<td>P6</td>
<td>AboutDialog</td>
<td>00:57 - 01:03</td>
<td>06</td>
</tr>
<tr>
<td>P6</td>
<td>MainFrame</td>
<td>02:40 - 02:57</td>
<td>17</td>
</tr>
<tr>
<td>P6</td>
<td>TextEditorPane</td>
<td>06:59 - 07:06</td>
<td>07</td>
</tr>
<tr>
<td>P6</td>
<td>Dialogs.registerDialog</td>
<td>10:50 - 10:56</td>
<td>06</td>
</tr>
</tbody>
</table>

Table 15.1: Instances of lag experienced by participants during Study 1

15.7.3 No support for reverse navigation

During relative navigation, a programmer navigates the call graph in a way that resembles a tree structure – starting from an initial method, each further decision regarding the method call to follow extends the branch of the tree that they are on. However, as it is usually possible for any method to call any other method in a program, each method in the tree navigated may also be approached by following a different path through the call graph. Before a programmer makes a modification to a method, it is often advisable to determine which other methods may
call it and determine whether making the planned change might cause unintended side effects. Therefore, a feature that has been regularly suggested is better support for reverse navigation back up such alternate call graphs branches. So far, implementing such functionality has been avoided due to the amount of rework required, as well as the inconsistencies it might bring to the Visuocode user interface. Currently, the flow view represents editor columns in such a way that columns to the right of an editor column are below it on the call graph. Potentially, each method editor might display a list of methods that call it, however, the question remains that if such a method were navigated to, where would it be represented? If an editor column were placed to the left of the called method, it would replace the existing method that called it – which would be undesirable. If an editor column were placed to the right of the called method, it would disrupt the consistency of the flow view. Potentially, it might be best if following such a link opened a new Flow window.

15.7.4 One flow view

For both studies, it was decided to use a build of Visuocode that only allowed one Flow window to be opened at a time to avoid the problem of window management. This may have affected the way that participants navigated as they may have been wary of the content of their flow window being lost when performing the next direct navigation. Several ways of supporting multiple flow views are currently being considered, including more intelligent positioning of flow windows when a new window appears, adding tabbing capability to the flow window so that it can hold multiple flow views, and allowing multiple flow views to be stacked within a flow window. It has been suggested the flow view might be collapsed to only show the names of the methods composing the flow view.

15.7.5 No support for search

The lack of textual search was identified during the expert feedback as potentially affecting the formal studies. This proved to be only an issue during the software navigation study as during the program composition study there was little need for search. It was observed during the participant sessions involving Eclipse that participants made more use of the Find function, which searches the current source code file for text, than the Search function, which searches the entire codebase. When a participant did make use of the Search function, it returned no results; they then resorted to using Find within the working set of files they had already made. It is concluded that searching for a text string within the current file is an important feature for an environment to support. Interestingly, while using Visuocode, textual search was often attempted as a proxy for being able to navigate back up the call graph, i.e., if the participant wanted to identify whether any methods of a class called a specific method.
15.8 Implications

The results of this project have the following implications for the continuing evolution of mainstream development environments.

15.8.1 Increased support for relative navigation

Development environments should provide additional support for relative navigation because relative navigation is a more natural style of navigation during the traversal stage of software understanding, and it decreases the likelihood of disorientation. Currently, if programmers traverse the call graph of software, they often become disoriented because the environment does not provide enough cues to allow them to retrace their steps. Additional support may be provided by either implementing a method-flow interface or by implementing functionality similar to Stackplorer [K+11], Blaze [KKK+13], or Prodet [AFQ+15].

15.8.2 Better representation for the composition hierarchy

Development environments should provide additional support for representing the composition hierarchy of software. Currently, most mainstream development environments focus on showing the programmer the directory and file structure of software. The programmer must then build a mental model of the composition hierarchy and interactions within the system with little support from the environment. Such support might be in the form of a visualisation that represents classes as boxes with the size of each box representing its size. A class that is composed of other classes would show those classes contained within it. Such a visualisation would allow the programmer to quickly determine the relationships between different classes.

15.8.3 Enhanced representation of class outlines

Development environments could provide an enhanced representation of the outline view of a class. Currently, while using Eclipse, programmers tend to open each class of interest and then scroll down through it in order to evaluate the number and size of methods it contains, as well as better understand its relationship to other classes. An expanded outline view might be developed that shows the members of a class, the size and complexity of methods, and the interaction between methods and members. Potentially, Eclipse could change the functioning of the Outline View so that it is also activated by selecting a class in the Package Explorer.
Chapter 16

Conclusions

The results of this project support the results of previous studies [KAM05, K11, KKK13] that identified that programmers often use a two-phase strategy while exploring unfamiliar code. First they carry out an exploration phase during which they opportunistically investigate source code files until they find a section of code that is potentially relevant – referred to as an anchor point. Next they begin a traversal phase during which they explore the call graph from the anchor point. Participants may perform many of these explorations and traversals while building up a working set of source code files relevant to the current task [RCM04].

It was found that while using Eclipse, during the exploration phase, participants would open class files and then scroll down through them in order to discover how they interrelated with other classes. The traversal phase would often involve opening classes that were members of previously viewed classes – it is conjectured that such navigation forms a mental-model of the software based on its composition hierarchy. This behaviour resulted in more direct and scrolling navigations being performed while using Eclipse than while using Visuocode. In contrast, while using Visuocode, during the exploration phase, participants would instead locate potential anchor points by first using the Workspace Manager window to open the outline of classes. During the traversal phase, participants used method-flow to navigate the call graph of the software, allowing them to discover the interrelationships between classes – it is conjectured that such navigation forms a mental-model of the software based on its call graph. This behaviour resulted in more relative navigations being performed while using Visuocode than while using Eclipse. Further research is required to determine the advantages and disadvantages of either form of mental model.

It is concluded that the method-flow visualisation technique is both intuitive to use and, importantly, also useful because all participants used method-flow extensively during at least one activity. Method-flow was used most extensively during the software navigation study to navigate code, and during such navigation no participants were observed to become disoriented, though identifying disorientation is admittedly quite subjective. Method-flow was mainly used to refer to Javadoc documentation during the program composition study.

Based on these results it is clear that mainstream development environments do not provide
adequate support for relative navigation. It was observed that while using Eclipse, participants would ‘try out’ relative navigation, then would revert to using direct navigation due to becoming disoriented. It is conjectured that, due to being name based, direct navigation causes a mental trail of breadcrumbs that can be more easily used to backtrack to previous source code locations. However, the results do suggest that if development environments did provide additional support for relative navigation – either via method-flow or some other mechanism – that programmers would use it.

In conclusion, due to the significant increase in the number of relative navigations performed by participants while navigating software using Visuocode than while using Eclipse, and due to the apparent difference in mental models formed while navigating using each environment, it is concluded that the method-flow visualisation technique implemented within Visuocode does provide additional support for relative software navigation. However, one issue with method-flow was identified. While using Visuocode, during traversal of the call graph, participants are not made aware of other methods in classes traversed – this is referred to as being ‘blinkered’. After their sessions, one participant expressed feeling misled by method-flow, and another mentioned that it took a while for them to get used to not looking at whole source files. While method-flow is considered to be a very promising technique for navigating the call graph, the effects caused by being ‘blinkered’ need to better understood.

### 16.1 Impact

During software understanding and program composition, programmers make use of various facilities provided by development environments to achieve their goals. Programmers use a mix of opportunistic search of the class hierarchy, and relative navigation along the call graph, to find a working set of files that need to be understood and/or modified in order to complete their tasks. It is clear that, of the user interfaces provided by recent prototype environments, no one interface alone provides a comprehensive solution for supporting programming, however, these interfaces in unison could increase programmer productivity.

The results of this study highlight that mainstream software development environments should provide explicit support for relative navigation. Participants were generally receptive of the method-flow concept, but when asked, were reticent to change environments, and indicated that they would prefer for a mechanism similar to method-flow to be incorporated into their preferred environment. Based on the reduction in total navigations performed while using Visuocode, and therefore assuming the time between navigations was more productive, there is a potential for an increase in programmer productivity if environments support method-flow.
Chapter 17

Future work

Future work can progress in the following areas.

A better understanding is needed of how the user interface presented by software development environments affects the mental model formed by a programmer during software understanding. In particular, it would be interesting to determine if other development environments that allow individual methods to be represented, such as Code Canvas, Code Bubbles, and Patchworks, also cause programmers to become ‘blinkered’.

While the empirical evaluation strategy of analysing video recordings of participant programming sessions in order to code their navigations was considered successful, the actual coding procedure was painstaking, laborious, and also error prone due to either losing concentration or becoming too interested in what the programmer was doing. During this project, there was an attempt to develop an instrumentation plug-in for Eclipse that would automatically record navigations, but it proved impossible given the Eclipse plug-in API available at that time. As Eclipse is open source, it may be worthwhile implementing such functionality by modifying Eclipse itself.

Further qualitative studies are needed to better understand how the number and type of navigations correlates with the success, or otherwise, of programming sessions. For example, a better understanding of whether the benefit of serendipitous discovery of code outweighs the disadvantage of time spent needlessly scrolling through source code.

In the future, it is planned to extend Visuocode to support additional programming languages, and also to support navigation between Java code and SQL Stored Procedures to better support the development of systems that rely on database functionality – similar to the work performed for the UQ Star development environment [WT92]. In particular, the editor columns within Visuocode may be enhanced to provide a better representation of whole classes so that programmers may better navigate the composition hierarchy using method-flow.
Appendix A

Study task sheets

This appendix includes recreations of the task sheets provided during participant sessions. Apart from correcting the name of the implementation class UniDateTime to DateTime, correcting a grammatical error ("and" to "a"), and correcting JavaChat to be described as a ‘chat program’, the following tasks sheets contain the same content as the original sheets distributed during participant sessions.
Task 1 – Text Editor++

TextEditor++ is an open source (GPLv2) text editor implemented in the Java Programming language. It is available at: http://jtexteditor.sourceforge.net

Each of the activities below will require you to use the provided programming environment for the purpose of better understanding the Text Editor++ application. You should not change the code of the application as you perform the tasks, however you may make notes within a text editor that is provided.

Each activity will ask you to investigate a specific aspect of the software. You may make notes as if you were intending to return to the software in the near future to make some changes. You should describe your impressions of how things work and interrelate.

Currently, the text editor does not support a number of functions that most programmers would consider necessary for a simple text editor. You are not asked to implement the following features, however you are asked to investigate the code with that goal in mind.

Activity 1 (10 minutes)

The text editor does not support syntax highlighting of text. Investigate the source code to determine how one might implement optional syntax highlighting, or reveal any impediments to such a feature.

Activity 2 (10 minutes)

Programmers often want to be able to indent (or de-indent) selected sections of text. Investigate the source code to determine how one might implement indenting of text, or reveal any impediments to such a feature.
Study 2 – Java Chat task sheet

Task 2 – JavaChat

JavaChat is an open source (GPLv2) chat program implemented in the Java Programming language. It is available at: http://java-chat.sourceforge.net

JavaChat includes both a chat client and a chat server. This task only requires you to navigate the source code of the client program, however the server code is also provided for completeness.

Each of the activities below will require you to use the provided programming environment for the purpose of better understanding the Java-Chat client application. You should not change the code of the application as you perform the tasks, however you may make notes within a text editor that is provided.

Each activity will ask you to investigate a specific aspect of the software. You may make notes as if you were intending to return to the software in the near future to make some changes. You should describe your impressions of how things work and interrelate.

While JavaChat is fairly complete and stable there are a couple of usability issues that might be addressed. You are not asked to implement the following features, however, you are asked to investigate the code with that goal in mind.

Activity 1 (10 minutes)

Currently, in JavaChat to connect to the server you need to register a new account, then you need to login. Explore the code and determine how you would go about automatically logging a person in after they have created an account.

Activity 2 (10 minutes)

Currently, when the JavaChat client starts the Login Dialog is automatically shown. This can be annoying if the user does not yet have an account.

Explore the code and determine how you might disable this automatically appearing. Then determine how you might modify the functionality so that the initial dialog allows the user to either login or create a new account.
Task: Telegram Report

Although telegrams are rarely used now, telegrams used to be a vital, but expensive, method of communication.

For this task, you are asked to implement a Java class that extends the ATelegramReport abstract class. Specifically, you need to provide an implementation of the “processTelegrams” abstract method, which is passed a Reader object that provides access to the telegram data, and returns a Vector of TelegramData objects.

Each telegram is separated by the sequence “ZZZZ”. The end of the stream is represented by an empty telegram, i.e., the last telegram is terminated by the string “ZZZZZZZZ”. The content only contains letters, numbers, and spaces. Individual words are separated by spaces, i.e. (’ ’).

Each TelegramData object corresponds to a telegram and contains the members “message”, “nrOfWords”, and “nrOfOverlengthWords”. A word is considered over length if it is greater than 12 characters in length. The sequences “STOP” and “ZZZZ” are not counted as words.

Composition

A skeleton for the class you are asked to implement is provided:

    org.controlledstudy.telegramreport.task.TelegramReport.java

Testing

A test class has been provided to test your implementation.
It is in the “org.controlledstudies.tests.telegramreport” package.

When using the command-line the tests can be activated by running:

    make run-tests
Study 2 – Date Time task sheet

Task: DateTime

Handling times and dates is an important, yet difficult task in most programming languages and environments. Firstly, times and dates are often provided as a text string in a variety of different formats. Secondly, each different programming language has a different set of relevant classes or functions for manipulating times and dates.

This task will have you implement a DateTime class that is able to be instantiated by passing in a string that may be in a variety of formats. As handling all formats would not be practical, only those specified in the IDateTime interface are asked to be implemented.

You may use any classes from the Java 7 standard class library, including the following classes that provide relevant functionality:

java.util.Calendar
http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Calendar.html

Note:
In Java 7, the java.util.Date class has been deprecated and is only used as an intermediary between parsing a date and creating a calendar.

```java
Date date = DateFormat.getInstance().parse( datetime );
Calendar cal = Calendar.getInstance().setDate( date );
```

The problem with this technique is that the formats handled by DataFormat are not well documented. It is suggested that you parse the date manually and use the relevant “set” methods on Calendar.

Important!

The "setMonth" method of Calendar takes an integer parameter where 0 corresponds to January, 1 to February, ..., 11 to December.

```java
Calendar cal = Calendar.getInstance();
cal.setMonth( 0 );
```

Testing

A test class has been provided to test your implementation. It is in the "org.controlledstudies.tests.datetime" package.
Appendix B

Study 1 participant summaries

B.1 Navigation summaries

This appendix provides a summary of each participant session. Each participant session is accompanied by a table containing a breakdown of the number and type of navigations performed. The first column identifies the participant, task, activity, and environment; the second group of columns represents direct navigations; the third group of columns represents relative navigations; the fourth group of columns represents scrolling navigations; and the last column totals all of the navigations. In these tables a dash, ‘-’, indicates that navigation type is not supported by the corresponding environment. The meaning of the symbols used for each column are described in Table 12.1. In the following descriptions ‘top-down’ refers to systematic investigation that follows the call graph, ‘opportunistic’ refers to opening files whose names appear related to the task, and ‘breadth-first’ refers to opening almost every file before any further systematic investigation.
Participant 1 (EVVE)

For both tasks, P1 used a top-down, opportunistic strategy. During Task 1, most time was spent scrolling through the `MainFrame` and `TextEditorPane` classes. At the start of the second activity, P1 was initially disoriented due to mis-remembering the name of the `TextEditorPane` class, and became confused as to why the Javadoc documentation was not what they expected. Eventually, P1 rediscovered the `TextEditorPane`, and was able to describe a solution based on the interaction between the `MainFrame` and `TextEditorPane` classes. During Task 2, P1 made extensive use of Visuocode’s flow view, however, during Activity 2 (while using Eclipse) P1 expressed surprise at finding the `doRegister` and `doLogin` methods within `ClientScreen` – while using Visuocode, these methods had not been shown because P1 had navigated by following a method call hyperlink, which would only show that one method.

Table B.1 shows the navigations performed by P1. Of the direct navigations, P1 performed more file navigations than the others combined. Of the relative navigations, P1 did not use the Eclipse Call Hierarchy navigation view, but did use relative navigation. Table B.2 shows the navigations performed by P1 grouped into direct, relative, and scrolling, navigations. P1 only used relative navigation three times while using Eclipse, but used it extensively while using Visuocode. In contrast, vertical scrolling was used extensively while using Eclipse, but not while using Visuocode. When grouped by kind, P1 performed more scrolling navigations than the other kinds combined, followed by relative navigations, then direct navigations.

<table>
<thead>
<tr>
<th>Session</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-T1-A1-E</td>
<td>3</td>
<td>1</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>P1-T1-A2-V</td>
<td>1</td>
<td>18</td>
<td>10</td>
<td>29</td>
</tr>
<tr>
<td>P1-T2-A1-V</td>
<td>7</td>
<td>21</td>
<td>14</td>
<td>42</td>
</tr>
<tr>
<td>P1-T2-A2-E</td>
<td>5</td>
<td>2</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>Total</td>
<td>16</td>
<td>42</td>
<td>113</td>
<td>171</td>
</tr>
</tbody>
</table>

Table B.1: Study 1: Navigations performed by P1

<table>
<thead>
<tr>
<th>Session</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-T1-A1-E</td>
<td>3</td>
<td>1</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>P1-T1-A2-V</td>
<td>1</td>
<td>18</td>
<td>10</td>
<td>29</td>
</tr>
<tr>
<td>P1-T2-A1-V</td>
<td>7</td>
<td>21</td>
<td>14</td>
<td>42</td>
</tr>
<tr>
<td>P1-T2-A2-E</td>
<td>5</td>
<td>2</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>Total</td>
<td>16</td>
<td>42</td>
<td>113</td>
<td>171</td>
</tr>
</tbody>
</table>

Table B.2: Study 1: Summary of navigations performed by P1
Participant 2 (EVVE)

For Task 1, P2 used a breadth-first strategy, which involved investigating a number of classes before focusing on the `TextEditorPane` class. P2, then, opportunistically scrolled through the `TextEditorPane` source file until coming to the `setFormat` method where they realised that the solution would involve working with the `JTextArea` class. During Task 2, P2 initially used a top-down strategy that led to the `ClientScreen` class. By expanding the `ClientScreen` class in the workspace window, P2 was able to see key methods such as `doRegister` and `doLogin`. During Task 2 Activity 2, P2 made extensive use of the flow view.

Table B.3 shows the navigations performed by P2. Like P1, P2 performed more file navigations than the other types of direct navigation. While using Eclipse, P2 made use of Eclipse’s ability to navigate back up the call hierarchy. Table B.4 shows the navigations performed by P2 grouped by kind. Similar to P1, the most number of navigations performed were scrolling navigations, followed by relative navigations, followed by direct navigations. The majority of scrolling navigations were performed while using Eclipse, while the majority of relative navigations were performed while using Visuocode.

<table>
<thead>
<tr>
<th>Session</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2-T1-A1-E</td>
<td>15</td>
<td>2</td>
<td>51</td>
<td>68</td>
</tr>
<tr>
<td>P2-T1-A2-V</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>P2-T2-A1-V</td>
<td>13</td>
<td>24</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>P2-T2-A2-E</td>
<td>6</td>
<td>13</td>
<td>17</td>
<td>36</td>
</tr>
<tr>
<td>Total</td>
<td>45</td>
<td>49</td>
<td>77</td>
<td>171</td>
</tr>
</tbody>
</table>

Table B.4: Study 1: Summary of navigations performed by P2
Participant 3 (EVVE)

During both tasks, P3 navigated the software in a goal-oriented fashion – focusing more on finishing the task rather than understanding the software. For Task 1, P3 appeared to use a breadth-first strategy that quickly led to the `TextEditorPane` class. However, once identified as the likely point of change, P3 needed further encouragement to continue investigation of how the change might be accomplished. For Task 2, P3 initially used a top-down strategy to find the `ClientScreen` class, then switched to a breadth-first strategy. In Visuocode, P3 quickly found the `doRegister` and `doLogin` methods by referencing the outline of the `ClientScreen` class in the Workspace Manager.

Table B.5 shows the navigations performed by P3. Like the previous participants, P3 performed more file navigations than the other direct navigations. In contrast to the previous participants, regardless of environment, P3 made little use of relative navigation. Table B.6 shows the navigations grouped by kind. Like P1 and P2, P3 performed more scrolling navigations than relative or direct, however, unlike P1 and P2, P3 performed more direct navigations than relative navigations.

![Table B.5: Study 1: Navigations performed by P3](image)

![Table B.6: Study 1: Summary of navigations performed by P3](image)
Participant 4 (VEEV)

For Task 1, P4 used a breadth-first strategy that quickly led to the `MainFrame` class and its `addDocument` method. This in turn led to the `JTextComponent` class (parent of `JTextArea`). P4 made extensive use of relative navigation using method-flow during Activity 1, and this style of navigation continued while using Eclipse during Activity 2. Interestingly, this behaviour disappeared when beginning the first activity of Task 2 while using Eclipse. This suggests that the use of method-flow navigation during Task 1 may have affected the way that P4 remembered the codebase, which later affected the navigation of the same codebase using Eclipse. During the last activity, while Visuocode was again used, method-flow navigation was again extensively used.

Table B.7 shows the navigations performed by P4. Unlike the previous participants, P4 used all forms of direct navigation except for search, however, file navigations were still used most. Relative navigation was used during all activities, while scrolling navigations were rarely used. Table B.8 shows the navigations performed by P4 grouped by kind. P4 performed more direct navigations than relative and scrolling navigations combined, and performed more relative navigations than scrolling navigations.

<table>
<thead>
<tr>
<th>Session</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4-T1-A1-V</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>P4-T1-A2-E</td>
<td>15</td>
<td>3</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>P4-T2-A1-E</td>
<td>17</td>
<td>6</td>
<td>6</td>
<td>29</td>
</tr>
<tr>
<td>P4-T2-A2-V</td>
<td>8</td>
<td>12</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>45</td>
<td>27</td>
<td>15</td>
<td>87</td>
</tr>
</tbody>
</table>

Table B.7: Study 1: Navigations performed by P4

<table>
<thead>
<tr>
<th>Session</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4-T1-A1-V</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>P4-T1-A2-E</td>
<td>15</td>
<td>3</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>P4-T2-A1-E</td>
<td>17</td>
<td>6</td>
<td>6</td>
<td>29</td>
</tr>
<tr>
<td>P4-T2-A2-V</td>
<td>8</td>
<td>12</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>45</td>
<td>27</td>
<td>15</td>
<td>87</td>
</tr>
</tbody>
</table>

Table B.8: Study 1: Summary of navigations performed by P4
Participant 5 (VEEV)

For both tasks, P5 initially used a top-down strategy. For Task 1, this quickly led to the `MainFrame` class, however, P5 then switched to a breadth-first strategy before returning to the `addDocument` method of the `MainFrame` class, which led to the `TextEditorPane` class. During Task 2, P5 used a more consistent top-down strategy, and made more use of relative navigation in both environments compared to during Task 1.

<table>
<thead>
<tr>
<th>Session</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5-T1-A1-V</td>
<td>14</td>
<td>8</td>
<td>39</td>
<td>61</td>
</tr>
<tr>
<td>P5-T1-A2-E</td>
<td>12</td>
<td>0</td>
<td>93</td>
<td>113</td>
</tr>
<tr>
<td>P5-T2-A1-E</td>
<td>5</td>
<td>0</td>
<td>44</td>
<td>86</td>
</tr>
<tr>
<td>P5-T2-A2-V</td>
<td>8</td>
<td>23</td>
<td>34</td>
<td>65</td>
</tr>
<tr>
<td>Total</td>
<td>72</td>
<td>43</td>
<td>210</td>
<td>325</td>
</tr>
</tbody>
</table>

Table B.10: Study 1: Summary of navigations performed by P5

Table B.9 shows the navigations performed by P5. Like the other participants, P5 performed more file navigations than the other direct navigations, however, P5 also performed a considerable number of search navigations. P5 performed relative navigations during all activities except for Task 1 Activity 2, while using Eclipse. Table B.10 shows the navigations performed by P5 grouped by kind. Like all other participants except P4, P5 performed more scrolling navigations than either direct or relative navigations. The second most navigations were direct navigations, then relative navigations.
Participant 6 (VEEV)

For both tasks, P6 used a breadth-first strategy and (like P3) carried out the activities in a goal-oriented fashion. More so than any of the other participants, P6 was disconcerted by Visuocode not showing Javadoc comments, as well by the few comments that were available using Eclipse. During Task 1, P6 made little use of Visuocode’s method-flow navigation, and no use of Eclipse’s relative navigation, however, P6 made extensive use of method-flow navigation later during Task 2.

Table B.11 shows the navigations performed by P6. Like the other participants, P6 performed more file navigations than any other form of direct navigation, however, while using Eclipse during Task 2 Activity 1, P6 performed a wide variety of different navigation types. P6 performed a variety of different relative navigations during all activities except Task 1 Activity 2, during which, no relative navigation were performed. Table B.12 shows the navigations performed by P6 grouped by kind. Like the other participants, P6 performed more scrolling navigations than the other kinds of navigation. The second highest number of navigations were direct navigations, then relative navigations.
B.2 Participant use of method-flow

This appendix describes how each participant made use of the method-flow functionality within Visuocode. P1, P2, and P3, attempted the TextEditor++ task activities in the order Eclipse, Visuocode, and the JavaChat task activities in the order Visuocode, Eclipse. P4, P5, and P6, attempted the TextEditor++ task activities in the order Visuocode, Eclipse; and the JavaChat task activities in the order Eclipse, Visuocode.

For typesetting purposes, in the descriptions below, package names are truncated to the name of the package that contains the relevant class. For example, \texttt{com.ays.javachat.client} is referred to as \texttt{client}. 

Use of method-flow by P1

TextEditor++ using Visuocode

P1 began the second activity of the TextEditor++ task by expanding the MainFrame class, navigating to it, and then scrolling down through it to the insertNewDocument method. From there, P1 first used method-flow to navigate to the JTabbedPane Javadoc, then scrolled between the insertNewDocument method and the Javadoc column viewer for several minutes. P1 then clicked on another method call, which caused the JScrollPane Javadoc to be shown. After scrolling down to the addDocument method, at 19:55, P1 followed a call to the TextEditorPane constructor, causing it to be opened in an adjacent editor column.

P1 used method-flow extensively to scroll back and forth between a method they were investigating and related Javadoc documentation. P1 also used method-flow to follow the method call from the addDocument method to the TextEditorPane constructor. However, while navigating using method-flow, it did appear that P1 was misled by the editor column only showing the TextEditorPane constructor, and not the other methods in that class.

JavaChat using Visuocode

P1 began the first activity of the JavaChat task by expanding the client package to reveal the Main class, then the manager package to reveal the ClientManager class. P1 then performed a direct navigation to the ClientManager class. P1 first used method-flow to navigate to the ClientScreen class from the ClientManager class. P1 then used the Workspace Manager to expand the screen package and the ClientScreen class, but continued navigating using the flow view by following the doConnect method from ClientScreen.start. P1 then navigated to a few classes using the Workspace Manager before navigating once more to the ClientManager class.

At 05:13, P1 began exploring the call graph of the ClientManager class by first using method-flow to navigate to the ClientTransmitter class, then navigated to the ClientScreen class. In particular, P1 used method-flow extensively to explore the call graph associated with the ClientScreen.start method.

Discussion

P1 adopted the strategy of clicking on an invoked object rather than the actual method call, which caused the entire class to be shown rather than just the single method. It is thought that this may have been to avoid being misled when an editor column only shows the single followed method. However, while exploring the methods called from the ClientScreen.show method, as the methods were implicitly called on the current class, P1 could only navigate to the single method.
Use of method-flow by P2

TextEditor++ using Visuocode

P2 began the second activity of the TextEditor++ task by expanding the TextEditorPane class and performing a direct navigation to the setKeyListener method. P2 then used method-flow to navigate to the KeyTextListener constructor, and then the TextEditorPane class (before scrolling down through it). Later at 14:09, P2 performed a direct navigation to the TextEditorPane constructor, then used method-flow to open the setKeyListener method again. Soon after, P2 followed the setText link of the TextEditorPane constructor to open the JTextComponent class in the flow view. Between 18:24 and 18:38, P2 investigated the KeyTextListener method using direct navigation from the Workspace Manager, but, at 18:38, then performed a direct navigation to the TextEditorPane.setKeyListener method, then used method-flow to navigate to the KeyTextListener constructor. At 19:12, P2 performed a direct navigation to the KeyTextListener class, which caused all of its methods to be displayed within the editor column. P2 spent the rest of the session using method-flow to explore from the KeyTextListener editor column – P2, first, navigated to the controlPress method, then replaced this editor column with the KeyEvent Javadoc by navigating to the getKeyCode method.

It appears that P2 used the Workspace Manager to find out information about class structure, then used method-flow to make sense of how the classes are related.

JavaChat using Visuocode

P2 began the first activity of the JavaChat task by performing a direct navigation to the Main class, then immediately used method-flow to navigate to the ClientManager constructor. P2 then directly navigated to ClientScreen.doConnect, then used method-flow to navigate to the ScreenCallback.connect method. Between 01:07 and 05:29, P2 continued this behaviour of directly navigating to a method in the ClientScreen class, then using method-flow to navigate to called methods. After performing a direct navigation to Main.main and using method-flow to navigate to the ClientManager constructor, between 05:30 and 08:34, P2 used method-flow to extensively explore the calls from the ClientScreen.start method. Later at 09:57, after performing a direct navigation to ClientScreen.doLogin, P2 used method-flow to navigate to sendRequest.

Discussion

P2 used method-flow to discover the relationships between classes, then used the Workspace Manager to see an overview of those classes. P2 then used method-flow to explore the call graph of relevant methods.
Use of method-flow by P3

TextEditor++ using Visuocode

During the second activity of the TextEditor++ task, P3 only used method-flow once to navigate from the TextEditorPane constructor to the JTextComponent Javadoc by following the setText link. However, it should be noted that P3 did not spend very long on this activity.

JavaChat using Visuocode

P3 began the first activity of the JavaChat task by expanding the client and interfaces packages, then performing a direct navigation to the Main class. Immediately after, P3 used method-flow to navigate to the ClientManager constructor, and then to the ClientScreen class. At 01:38, P3 performed a direct navigation to the ClientScreen.doRegister method, then used method-flow to open the LoginData class.

Discussion

P3 mainly used method-flow to investigate the call graph from the main and doRegister methods. In particular, method-flow allowed P3 to quickly discover the importance of the ClientScreen class, which then allowed them to discover the doRegister method using the class outline in the Workspace Manager. After navigating to doRegister, method-flow allowed them to quickly open the LoginData class in an editor column.
Use of method-flow by P4

TextEditor++ using Visuocode

P4 began the first activity of the TextEditor++ task by first opening the Page class, then opening the MainFrame class. P4 then used method-flow to navigate to the addDocument method of the MainFrame class, before navigating directly to the Page class again. Discounting the Page class, P4 then navigated back to the MainFrame.openDocument method, then used method-flow to navigate, again, to addDocument. From addDocument, P4 then used method-flow to follow the call to the TextEditorPane.setText method, which opened the JTextComponent Javadoc in an editor column. For the rest of the session, P4 navigated, scrolling back and forth, between the editor columns within the flow view.

JavaChat using Visuocode

P4 began the second activity of the JavaChat task by navigating to the ClientScreen class, which revealed the ClientScreen constructor, as well as several other methods in the class including the start method. Seeing the call to doConnect within the start method, P4 then used method-flow to open that method. From doConnect, P4 then followed the screenCallback.connect method call, but this resulted in the ScreenCallback interface being opened, not the connect method of the class implementing ScreenCallback. After some effort, P4 determined that the ClientManager class implemented ScreenCallback and was passed to the ClientScreen class. This led to a direct navigation to the ClientManager class, then a relative navigation, using method-flow, to the replayReceived method of ClientScreen. Soon after, P4 performed a direct navigation to the ClientScreen class, and spent the remainder of the session exploring its methods using method-flow.

Discussion

It appears that P4 used method-flow to discover the relationships between classes, then performed a direct navigation to understand the members and methods of a specific class. Method-flow allowed P4 to explore method calls with very little penalty if the navigation was fruitless. Using method-flow, P4 was able to quickly identify the main classes of each program.
Use of method-flow by P5

TextEditor++ using Visuocode

P5 began the first activity of the TextEditor++ task by opening the MainFrame class, then followed the method call to setTitle using method-flow, which caused the Frame class Javadoc to be opened. P5 then directly navigated to the TextEditor.main method, then navigated to the MainFrame constructor. P5 then re-performed the direct navigation to the MainFrame class, and proceeded to scroll down through its methods. After navigating to the KeyTextListener class, P5 used method-flow to navigate to the controlPress method (even though it was already on screen). After navigating to the MainFrame.readFTFFile method, P5 used method-flow to navigate to the ObjectInputStream Javadoc. Then, after navigating to the MainFrame.openDocument method, P5 used method-flow to navigate to the addDocument method. After performing a direct navigation to the TextEditorPane class, P5 spent the remainder of session scrolling through its methods. While juxtaposing the openDocument and addDocument methods, P5 adjusted the width of the openDocument editor column to fit both methods on screen.

JavaChat using Visuocode

P5 began the second activity of the JavaChat task by opening the ClientManager class, then following the clientScreen.start method call in the constructor. P5 then tried to use method-flow to open the ClientScreen class, but because of lag (because it is a large class), followed the start method just as the class appeared. P5 then reopened the ClientScreen class, but decided to navigate by method instead, and re-followed the start method, then followed the doConnect method. Soon after, P5 performed a direct navigation to open the ClientScreen class, then opened its outline. P5 then investigated the ClientScreen class by first scrolling down through it, then by selecting individual methods. For the rest of the session, P5 used method-flow to investigate the relationship between the ClientScreen.replyReceived method and the methods it calls.

Discussion

P5 used the Workspace Manager to understand the structure of classes, then method-flow to better understand the relationships between classes. P5 also made use of the class members area that is located in the top area of the column editor – as the members area can be scrolled it doesn’t affect the methods area.
Use of method-flow by P6

TextEditor++ using Visuocode

P6 began the first activity of the TextEditor++ task by experimenting with the interface by opening the AboutDialog class. P6 then opened the TextEditor.main method, and used method-flow to open the MainFrame.addDocument method. Unfortunately, due to a bug, the addDocument method call did not properly resolve, which caused a method skeleton to appear instead of the actual method. However, P6 then immediately followed a link to the MainFrame class, causing it to open in the adjacent column editor. After using the Workspace Manager to investigate the methods of the other classes, P6 focused on searching through the methods of the TextEditorPane class using direct navigation.

JavaChat using Visuocode

During the second activity of the JavaChat task, after being given a hint regarding how dialogs block execution in Java, P6 used method-flow extensively to investigate the methods called from ClientScreen.actionPerformed.

Discussion

P6 appeared to use the Workspace Manager to investigate what each class did, in order to find appropriate classes to investigate, but then made extensive use of method-flow to understand the call graph from these classes.
Appendix C

Study 2 participant summaries

C.1 Navigation summaries

For each participant, this appendix provides a brief description of how they attempted each task, and also discusses how they navigated. For each participant, a table is provided that shows how many of each navigation type they performed.
Participant 1

Task 1 – Telegram Report using Eclipse

P1 began the Telegram Report task by investigating the *TelegramReport* abstract class and *TelegramData* class. While using Eclipse, P1 used a strategy whereby they dragged source code files that would be referenced to the right-hand side of the Eclipse window, which allowed those files to be visible within an adjacent editor. At 04:11, P1 began modifying the *TelegramReport.processTelegrams* method, then began implementing their own *Scanner* class. During this session, P1 created new software structure by using Eclipse’s auto-create functionality. While implementing the *Scanner* class, P1 dragged the *TelegramReport* class to a right-hand editor tab set, then when completed (at 21:33) moved it back to the left-hand editor tab set. At 33:45, as P1 could not think of anything left to do, the experimenter suggested they run the tests. For the remainder of the session, P1 continued to fix miscellaneous errors.

Task 2 – Date Time using Visuocode

Similar to the previous session, P1 began the Date Time task by investigating the provided classes. At 06:21, P1 began the modification phase by adding several members to the *Date-Time* class. While implementing the *DateTime.parse* method, P1 used Visuocode’s auto-create functionality to create the new methods in an adjacent column editor. During the rest of the session, P1 continued to implement these methods.

Discussion

Due to using the strategy of juxtaposing editors on each side of the Eclipse window, and using method-flow while using Visuocode, P1 performed more local and lateral navigations (→, ←, and →) than direct navigations (⊕, ⊖, and ⊗), however the number of local navigations was considerably higher during Task A while using Eclipse. P1 performed very few scrolling

<table>
<thead>
<tr>
<th>Session</th>
<th>→</th>
<th>←</th>
<th>→</th>
<th>⊕</th>
<th>⊖</th>
<th>⊗</th>
<th>←</th>
<th>←</th>
<th>→</th>
<th>0</th>
<th>⊖</th>
<th>↑</th>
<th>↓</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-E-TG</td>
<td>37</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>P1-V-DT</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11</td>
<td>13</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>44</td>
<td>9</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>11</td>
<td>13</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Mean</td>
<td>22.00</td>
<td>4.50</td>
<td>4.50</td>
<td>8.00</td>
<td>2.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>5.50</td>
<td>6.50</td>
<td>2.50</td>
<td>0.50</td>
<td>0.50</td>
<td>0.00</td>
</tr>
<tr>
<td>Range</td>
<td>30.00</td>
<td>1.00</td>
<td>1.00</td>
<td>2.00</td>
<td>4.00</td>
<td>0.00</td>
<td>2.00</td>
<td>2.00</td>
<td>11.00</td>
<td>13.00</td>
<td>5.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table C.1: Study 2: Navigations performed by P1

<table>
<thead>
<tr>
<th>Session</th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Editing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-E-TG</td>
<td>45</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>35</td>
<td>96</td>
</tr>
<tr>
<td>P1-V-DT</td>
<td>17</td>
<td>9</td>
<td>29</td>
<td>2</td>
<td>13</td>
<td>70</td>
</tr>
<tr>
<td>Total</td>
<td>62</td>
<td>24</td>
<td>29</td>
<td>3</td>
<td>48</td>
<td>166</td>
</tr>
</tbody>
</table>

Table C.2: Study 2: Summary of navigations performed by P1

220
navigations during either task. While using Eclipse during Task 1, P1 initiated more than double the number of edits ($\Delta$) than while using Visuocode during Task 2.
Participant 2

Task 1 – Date Time using Eclipse

P2 began Task 1 by first investigating the TestDateTime class, then the ADateTime abstract class and DateTime class. At 07:07, P2 began modifying the DateTime.parse method. At first P2 considered using regular expressions, but then reconsidered and decided to use a simpler divide and conquer approach. Similar to P1, P2 used Eclipse’s auto-create functionality to create two methods – DateTime.splitDate and DateTime.splitTime – however these were never used as it was easier to use the String arrays returned from the String.split method. At 39:01, P2 attempted to use the Eclipse refactoring functionality to extract a section of text into a method, however the attempt had to be aborted and retried (potentially due to the refactoring dialog not being easy to use). At 53:56, P2 ran the tests for the first time leading to the realisation that their solution did not properly handle strings containing only dates. The rest of the session, until 60:43, was spent debugging.

Task 2 – Telegram Report using Visuocode

P2 began Task 2 by investigating the TelegramData and TelegramReport classes, as well as studying the Reader Javadoc by using method-flow to open the Javadoc in an adjacent view in the flow view. At 04:46, P2 opened a Chrome web browser to refer to the Reader Javadoc, and later the Scanner Javadoc. During the modification phase, P2 mainly swapped back and forth between the TelegramReport.processTelegrams method and the Javadoc documentation in the web browser. At 19:50, P2 ran the tests to evaluate their solution for the first time, and after fixing several issues, at 21:27, ran the tests again which passed.

<table>
<thead>
<tr>
<th>Session</th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Editing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2-E-DT</td>
<td>19</td>
<td>74</td>
<td>6</td>
<td>16</td>
<td>42</td>
<td>157</td>
</tr>
<tr>
<td>P2-V-TG</td>
<td>21</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>45</td>
</tr>
<tr>
<td>Total</td>
<td>40</td>
<td>83</td>
<td>9</td>
<td>16</td>
<td>54</td>
<td>202</td>
</tr>
</tbody>
</table>

Table C.3: Study 2: Navigations performed by P2

Table C.4: Study 2: Summary of navigations performed by P2
Discussion

P2 completed Task 2 in less than half the amount of time spent on Task 1. During the first task (Date Time), P2 predominantly performed tabbing navigations, while the number of local and direct navigations – less tabbing navigations – were near equal. In contrast, during the second task (TelegramReport), P2 mainly used lateral navigations (← and →) and direct navigations using the Visuocode Workspace Manager (⊕). During both tasks, P2 made use of forward relative navigation to view related Javadoc; and, during Task 1, to create called methods. During Task 2, P2 initiated slightly more than a quarter of the number of edits, and performed less than a third of the number navigations, compared to Task 1.
Participant 3

Task 1 – Telegram Report using Eclipse

P3 began Task 1 by investigating the provided classes, then navigating to (and reading) the Eclipse provided `Reader` class. At 06:45, they then investigated and ran the tests. P3 then opened a browser and searched for the Javadoc for the `Reader` class. At 15:16, P3 began modifying code by importing the `java.util.Scanner` class, then incorporating its use into the `TelegramReport.parseTelegrams` method. While editing, P3 would periodically swap back and forth between the Eclipse and web browser windows, though eventually P3 resized the Eclipse window so that both windows could be seen simultaneously. At 25:19, P3 opened the `TelegramData` class in another tab, then periodically swapped back and forth between the `TelegramReport` and `TelegramData` tabs while completing the method. At 47:12, P3 began the debugging phase by running the tests, which passed several minutes later.

Task 2 – Date Time using Visuocode

P3 began Task 2 by reading through the `ADateTime` abstract class and the `DateTime` class. At 04:08, P3 began modifying the code by creating a `Calendar` member within the `DateTime` class, then proceeded to open the `Calendar` Javadoc in a browser window. P3 then began implementing the `DateTime` accessor methods first (contrasting with the other participants who implemented them last). At 24:19, P3 began implementing the `DateTime.parse` method – periodically referring to the `String.split` Javadoc. At 60:09, P3 began debugging their code and the rest of the session was spent debugging. At 81:00, it was agreed to finish the session.

<table>
<thead>
<tr>
<th>Session</th>
<th>←→</th>
<th>←→</th>
<th>↓↑</th>
<th>←→</th>
<th>↑↓</th>
<th>Δ</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3-E-TG</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>P3-V-DT</td>
<td>36</td>
<td>53</td>
<td>57</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>37</td>
</tr>
<tr>
<td>Total</td>
<td>45</td>
<td>66</td>
<td>71</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>87</td>
</tr>
</tbody>
</table>

Mean: 22.50 33.00 35.50 4.50 1.50 0.00 4.00 1.50 16.00 17.00 33.50 180.50
Median: 22.50 33.00 35.50 4.50 1.50 0.00 4.00 1.50 16.00 17.00 33.50 180.50
Range: 27.00 40.00 43.00 1.00 3.00 0.00 9.00 9.00 1.00 2.00 0.00 6.00 1.00 2.00 0.00 27.00 115.00

Table C.5: Study 2: Navigations performed by P3

<table>
<thead>
<tr>
<th>Session</th>
<th>Implicit</th>
<th>Direct</th>
<th>Relative</th>
<th>Scrolling</th>
<th>Editing</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3-E-TG</td>
<td>36</td>
<td>26</td>
<td>1</td>
<td>40</td>
<td>20</td>
<td>123</td>
</tr>
<tr>
<td>P3-V-DT</td>
<td>146</td>
<td>4</td>
<td>4</td>
<td>37</td>
<td>47</td>
<td>238</td>
</tr>
<tr>
<td>Total</td>
<td>182</td>
<td>30</td>
<td>5</td>
<td>77</td>
<td>67</td>
<td>361</td>
</tr>
</tbody>
</table>

Table C.6: Study 2: Summary of navigations performed by P3
Discussion

P3 performed less than a third the number of local navigations (\(\sim\)) during Task 1 than during Task 2. This appears due to the longer duration of Task 2, and also the presence in the second task of the \emph{DateTime} class, which increases the likelihood of local navigations due to having a lot of short accessor methods. During both tasks, P3 performed numerous lateral navigations (\(\leftarrow\) / \(\rightarrow\)) to swap back and forth between the development environment window and either a browser window or the command-line – however, four times as many were performed while using Visuocode. P3 performed relatively few direct navigations (\(\oplus\)/\(\ominus\)). While using Eclipse, P3 would tab between open files, and while using Visuocode, a large amount of time was spent scrolling down through the \emph{DateTime} class while implementing its accessor methods. During Task 1, P3 initiated roughly half the number of edits, and performed half the number of navigations, compared to Task 2.
Participant 4

Task 1 – Date Time using Eclipse

P4 began the first task by investigating the provided classes, then searching for and reading the `java.util.Calendar` Javadoc. At 12:53, P4 created the `DateTime.calendar` member, then began altering the `DateTime.parse` method. Initially, P4 considered processing the date using regular expressions, and proceeded to swap back and forth between Eclipse and the Java Pattern Javadoc. At 23:05, the experimenter suggested splitting between the date and time using the `String.split` method. For the remainder of the session, P4 continued to modify the `DateTime.parse` method while periodically referring to the `ADateTime` class (which was placed in the right-hand side of the Eclipse window). At 61:21, the session ended.

Task 2 – Telegram Report using Visuocode

P4 began the second task by perusing the `TelegramReport.processTelegrams` method, and then the `Reader` Javadoc within the flow view. Between 03:49 and 15:50, P4 implemented the `processTelegrams` method (swapping to the `ATelegramReport` Xcode window once). At 15:50, P4 commenced debugging by running the tests, and, at 17:42, the tests passed.

Discussion

P4 performed half the number of local navigations (\(\sim\)) during Task 2, but 5 times as many direct navigations (\(\oplus\)). This appears due to Visuocode encouraging navigation using individual methods, which reduces the chance for serendipitous navigation between methods in a file. P4 performed many more lateral navigations (\(\leftarrow / \rightarrow\)) during Task 1 due to first needing to refer to the Pattern Javadoc, then later needing to refer to the `ADateTime` class. P4 did not use relative navigation (\(\Rightarrow, \Leftarrow, \text{and} \rightarrow\)) while using Eclipse, but did use it while using Visuocode to lookup Javadoc documentation. During Task 1, P4 performed very few scrolling navigations, and during Task 2, no scrolling navigations were performed. P4 initiated double the number of edits and performed roughly double the number navigations during Task 1 than during Task 2 – likely due to the duration of Task 1 being double that of Task 2.

<table>
<thead>
<tr>
<th>Sessions</th>
<th>(\sim)</th>
<th>(\leftarrow)</th>
<th>(\rightarrow)</th>
<th>(\oplus)</th>
<th>(\leftarrow)</th>
<th>(\rightarrow)</th>
<th>(\uparrow)</th>
<th>(\downarrow)</th>
<th>(\Delta)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4-E-DT</td>
<td>7</td>
<td>23</td>
<td>21</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P4-V-TG</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>10</td>
<td>25</td>
<td>27</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Mean</td>
<td>5.00</td>
<td>12.50</td>
<td>13.50</td>
<td>6.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.50</td>
<td>1.00</td>
<td>1.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Median</td>
<td>5.00</td>
<td>12.50</td>
<td>13.50</td>
<td>6.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.50</td>
<td>1.00</td>
<td>1.50</td>
<td>3.00</td>
</tr>
<tr>
<td>Range</td>
<td>4.00</td>
<td>19.00</td>
<td>15.00</td>
<td>6.00</td>
<td>0.00</td>
<td>0.00</td>
<td>3.00</td>
<td>2.00</td>
<td>3.00</td>
<td>6.00</td>
</tr>
</tbody>
</table>

Table C.7: Study 2: Navigations performed by P4

226
Table C.8: Study 2: Summary of navigations performed by P4

Participant 5

Task 1 – Date Time using Eclipse

P5 began the first task by exploring the package hierarchy in the Eclipse Package Explorer, then investigating the ADateTime abstract class and DateTime class. At 08:45, due to the despondency of the participant, as they were not familiar with the Java class library, the experimenter suggested looking at the tests to better understand how the DateTime object is used. At 19:18, P5 began modifying the DateTime.parse method. Due to previous prompting by the experimenter, P5 used a divide and conquer strategy of replacing the ‘T’ in the date with a space, and then splitting on spaces to retrieve individual date and time components. At 43:12, P5 created the get date parts method, which takes a date string and returns an array of integers. At 51:52, the session ended due to running out of time.

Task 2 – Telegram Report using Visuocode

P5 began Task 2 by looking through the ATelegramReport and TelegramData classes, then later opening the Reader Javadoc within the flow view. At 12:26, P5 began modifying the TelegramReport.processTelegrams method – using a strategy of processing the characters from the Reader within the single method. At 27:47, P5 ran the tests for the first time and spent the rest of the session debugging their code. At 50:18, as P5’s code passed four of the five tests, but did not correctly detect the final empty telegram, it was decided to finish the session.

Table C.9: Study 2: Navigations performed by P5

Table C.10: Study 2: Summary of navigations performed by P5
Discussion

P5 performed more local navigations and fewer direct navigations during Task 1 while using Eclipse, and fewer local navigations and more direct navigations during Task 2 while using Visuocode. While using Eclipse, P5 made extensive use of tabbing to swap back and forth between the `DateTime` and `ADateTime` classes. Relative navigation was only used once, while using Visuocode, to open the Reader Javadoc in the flow view. During Task 1, while using Eclipse, scrolling was extensively used, however, it was not used at all while using Visuocode during Task 2. P5 initiated fewer edits while using Eclipse than while using Visuocode, but performed nearly double the number of navigations.
C.2 Participant use of method-flow

This appendix describes how each participant used method-flow during the task they used Visuocode. P1 and P3 used Visuocode while attempting the Date Time task, and P2, P4, and P5 used Visuocode while attempting the Telegram Report task.

Use of method-flow by P1

At the beginning of the Date Time task, P1 used direct navigation via the Workspace Manager to navigate between and investigate the provided classes. P1’s first use of method-flow was to reveal the Javadoc documentation for the String.split method. Soon after, P1 used method-flow to create the DateTime.parseTime method by writing a call to the non-existent method, and then using Visuocode’s auto-create functionality to display a method skeleton in an adjacent editor column within the flow view. From the DateTime.parseTime method, P1 used method-flow again to reveal the String.split Javadoc again. P1 then scrolled back to the left to continue editing the DateTime.parse method. Soon after, P1 used method-flow to create the DateTime.parseISODate method. While attempting this task, the main benefit of method-flow to P1 was the ability to have the DateTime.parse method constantly juxtaposed with the methods it was calling while they were being implemented. Additionally, while creating the new methods, the layout of the column editor allowed P1 easy access to the imports and members lists of the class.

Use of method-flow by P2

At the beginning of the Telegram Report task, P2 used direct navigation via the Workspace Manager to navigate between, and investigate, the provided classes. P2 first used method-flow to look at the Javadoc documentation for the Reader class, however, they soon opened this in a web browser instead. P2 only used method-flow to create new code once due to accidentally creating a new TelegramData constructor because it was assumed that a blue-coloured hyperlink-enabled constructor call would navigate to the already existing constructor. This was caused by a design issue with Visuocode – non-existent constructor calls should be coloured red, not blue. It does highlight, however, that Visuocode should prompt the user regarding the arguments accepted by existent methods and constructors.

Use of method-flow by P3

At the beginning of the Date Time task, P3 first opened a web browser to investigate the Javadoc documentation for the Calendar class. P3 then proceeded to scroll down through the DateTime class implementing the accessor methods. P3 later returned to complete the DateTime.parse method. At no time during the task did P3 use method-flow. By accident,
when P3 started this task, a flow window containing the *DateTime* class was already open – this may potentially have affected the way that P3 used the Visuocode environment.

**Use of method-flow by P4**

At the beginning of the Telegram Report task, P4 used direct navigation via the Workspace Manager to navigate between, and investigate, the provided classes. Like P2, P4 first used method-flow to look at the Javadoc documentation for the *Reader* class. Then later, after almost completely implementing the `processTelegrams` method, P4 used method-flow again to reference the *String* Javadoc documentation.

**Use of method-flow by P5**

At the beginning of the Telegram Report task, P5 used direct navigation via the Workspace Manager to navigate between, and investigate, the provided classes. Like P2 and P4, P5 first used method-flow to look at the Javadoc documentation for the *Reader* class. P5 then didn’t use method-flow until after the task had finished – while completing the questionnaire, they navigated to the *TelegramData* class.

**Summary**

Every participant except P3 spent a short amount of time performing direct navigations using the Workspace Manager in order to orient themselves to the provided source code. Every participant except P3 then used method-flow to open the Javadoc documentation for a relevant class in the flow view, and all of these participants continued to use method-flow in this way except P2 who opened a web browser. While using Visuocode, P1 was the only participant to seriously use method-flow for developing new code as the other participants attempted to implement their solutions using the existing stub method.
Bibliography


A. Begel. LogoBlocks: a graphical programming language for interacting with the world. 1996.


Index

algorithm animation, 46
analysis method, 127, 134
anchor point, 21, 135, 151, 186, 197
automatic programming, 55
between-subjects, 133
call graph, 20
class attributes area, 113
class browser, 62, 108
code augmentation environment, 57, 59
codespace, 42, 43, 67, 186, 187
cognitive map, 22, 35, 42, 186
cognitive model, 39, 41
companion editors, 102
companion visualisations, 51
composition hierarchy, 20, 99, 158, 196, 197
control structure diagram, 51, 59, 102
counter-balanced, 127
data, 127
dependent variables, 127, 129, 144, 166
direct navigation, 20, 76, 130
editing behaviour, 132
editor column, 25, 28, 95
emergent structure, 20
enhanced representation, 51
ethical clearance, 126
execution visualisation, 46, 48
experienced programmers, 128
experiment plan, 28, 127
experimental design, 127, 133, 144, 166
external cognition, 37
external representation, 32, 37, 43
flow view, 25, 94
Flow window, 95, 113
human problem solving, 39, 39
hyperspace, 20, 93
hypothesis, 26, 27
implicit navigation, 21, 77, 129
inheritance hierarchy, 20
integrated development environment (IDE), 54, 64
keyhole phenomenon, 22
landmarks, 23, 42, 84
layers of abstraction, 28, 99
learning effect, 133, 166
literate programming, 58
long-term memory, 23, 32
mental imagery, 36
mental model, 34, 41
meta visualisation, 46, 49
method-flow visualisation technique, 25, 95
methods area, 113
multi-view systems, 55
navigational disorientation, 22
participants, 22, 127, 128
polymorphism, 21, 101
program composition, 23, 96, 115
program comprehension, 35, 41, 96
program state visualisation, 46, 48
program visualisation, 43, 45, 46
programming artefacts, 77, 81, 82
programming context, 23, 95, 100
programming environments, 54
protocol analysis, 127
psychology of computer programming, 39
qualitative data, 127, 129
recommender systems, 56, 58
refactoring, 162
relative navigation, 21, 57, 76, 131
relative navigation environments, 56, 57
repeated measures t-test, 134, 150, 175
scrolling navigation, 21, 76, 131
semantic knowledge, 35, 40
session transcript, 129
short-term memory, 32
software ageing, 19
Software Dimensions, 28, 98
software engineering, 19
software exploration, 22, 51
software navigation, 20, 28
software psychology, 39
software quality, 19
software understanding, 26, 28, 51
software visualisation, 26, 44, 45
spatial consistency, 26, 81
spatial memory, 22, 25, 33, 35, 57
spatial navigation environments, 56, 57, 119
spatial separation, 81, 82
spatially consistent environment, 81, 83
static analysis, 117
syntactic knowledge, 40
syntax-directed editors, 55
task context, 58
tasks, 128, 142, 165
temporality, 84
thrashing, 57, 96
training task, 133
traversal phase, 21, 135, 197
treatments, 127, 128
two-phase strategy, 21, 135, 197
visual momentum, 22
visual programming, 42, 44, 45
visualisation, 44
visuo-spatial environment, 78, 119
visuo-spatial memory, 25, 78
visuo-spatial programming, 25
visuo-spatial programming interface, 28
visuo-spatial reasoning, 37
visuo-spatial sketchpad, 25, 33, 78
Visuocode, 25, 112
within-subjects, 127, 133
working memory, 24, 32, 78
working set, 21, 58
working set environments, 56, 58
Workspace Manager, 112