Onboarding in Software Engineering

Rebecca Yolande Yates

Thesis presented for the degree of
Ph.D. in Software Engineering
to the
Computer Science and Information Systems Department
Faculty of Science & Engineering
University of Limerick

Research Supervisors
Dr. Jim Buckley
and
Dr. Norah Power

Submitted to the University of Limerick
November 2014
Abstract

Onboarding in Software Engineering
Rebecca Yates

Many software developers struggle to understand code written by others, leading to increased maintenance costs. Research on program comprehension to date has primarily focused on individual developers attempting to understand code. However, software developers also work together to transfer understanding of their codebases. This activity is common during the onboarding process, when a new developer has joined a project.

This study uses a Grounded Theory approach to explore the information passed from expert to newcomer, the techniques used during onboarding sessions and the value of this information to the newcomer. The theory is grounded in recordings of twelve onboarding sessions, supplemented with interviews and questionnaires, and answers questions about the representation of code, the support offered to newcomers and the problems encountered during the onboarding process.

In addition to providing a focused study of the content of onboarding sessions, this study reveals some novel aspects of software comprehension (in particular the need for the Temporal and Rationale views of the code in addition to Structural and Algorithmic) and provides a set of recommendations to increase the effectiveness of onboarding sessions.

The theory is evaluated for both fit and generalisability, demonstrating its applicability to industry, and linked to previous work on software comprehension, concept and feature location, information seeking, information push and pull, and onboarding.
Declaration

The work described in this thesis is, except where otherwise stated, entirely that of the author and has not been submitted in any part for a degree at this or any other University.

Signed:

Rebecca Yolande Yates

Date:
Acknowledgements

Firstly, thank you to my supervisors, Jim Buckley and Norah Power, for your guidance and endless encouragement. Occasionally you even agreed on something, and every time we met you renewed my enthusiasm for the work ahead. Thank you both!

Many thanks also to:

- All the anonymous developers, for your participation, and to everyone who helped to organise the case studies - this research would not have been possible without you.
- Chris Exton and Maria Kutar, for examining the thesis and for your feedback afterwards.
- My friends and colleagues at Lero and CSIS, especially Mike Hinchey.
- Colm McGettrick, for rescuing a recording from technological disaster.
- Howell Jordan, for tea and tangents.
- The PPIG community, for the warm welcome you extend to students, and especially Thomas Green, Maria Kutar, Marian Petre and Luke Church, for your friendship and wisdom.

Finally, love and thanks to my parents, Yo and Keith, and to Charlie, for all your support over the years.

This work was supported, in part, by Science Foundation Ireland grants 03/CE2/I303_1 and 04/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).
Contents

Abstract iii
Declaration v
Acknowledgements vii
Contents ix
List of Tables xv
List of Figures xviii
Taxonomy of Terms xix

1 Introduction 1
  1.1 Research Questions . . . . . . . . . . . . . . . . . . . . . . 3
  1.2 Scope of this study . . . . . . . . . . . . . . . . . . . . . . 3
  1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . 4
  1.4 Thesis plan . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 Literature Review 7
  2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Program Comprehension . . . . . . . . . . . . . . . . . . . 8
    2.2.1 Beacons . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.2.2 Plans . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.2.3 Mental models . . . . . . . . . . . . . . . . . . . . . 10
    2.2.4 Top-down, bottom-up and the influence of domain
        knowledge . . . . . . . . . . . . . . . . . . . . . . . 11
    2.2.5 Integrated models of comprehension . . . . . . . . . . 12
    2.2.6 Influence of the task on comprehension strategy . . . 13
    2.2.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . 14
  2.3 Information seeking . . . . . . . . . . . . . . . . . . . . . 17
    2.3.1 Models of information seeking . . . . . . . . . . . . . 17
    2.3.2 Information sources . . . . . . . . . . . . . . . . . . . 20
    2.3.3 Information needs . . . . . . . . . . . . . . . . . . . . 22
    2.3.4 Tool support for information seeking . . . . . . . . . 23
## CONTENTS

2.3.5 Summary ............................................. 25
2.4 Concept and feature location ................................. 26
  2.4.1 Techniques and issues .............................. 27
  2.4.2 Summary .......................................... 29
2.5 Push and pull ........................................... 29
  2.5.1 Studies of push .................................. 31
  2.5.2 Summary .......................................... 33
2.6 Onboarding .................................................. 33
  2.6.1 Newcomer issues ................................. 35
  2.6.2 Experts as mentors ............................... 37
  2.6.3 Summary .......................................... 39
2.7 Discussion ............................................... 39

3 Methodology ................................................ 41
  3.1 Quantitative and Qualitative Research ......................... 41
  3.2 Methodologies for exploratory studies .......................... 43
  3.3 Techniques for exploratory studies .......................... 45
  3.4 Methodology for this study ................................ 47
    3.4.1 The Grounded Theory methodology .................. 47
  3.5 Summary ............................................... 53

4 Research Design ........................................... 55
  4.1 Pilot studies ......................................... 55
  4.2 Finding participants .................................. 57
  4.3 Data collection ...................................... 58
    4.3.1 Recording the onboarding session .................. 58
    4.3.2 Questionnaires .................................. 59
    4.3.3 Followup interviews ............................. 60
  4.4 Measuring software development skill ....................... 61
  4.5 Preparing data ....................................... 62
  4.6 Tool support for analysis ................................ 62
  4.7 Analysis ............................................. 63
  4.8 Evaluation ........................................... 66
  4.9 Summary ............................................... 67

5 Empirical Study ........................................... 69
  5.1 Characterisation of cases ................................ 69
  5.2 Variations in sessions ................................ 72
    5.2.1 People present .................................. 72
    5.2.2 Project context .................................. 73
    5.2.3 Session setup ................................... 73
    5.2.4 Onboarding aspects .............................. 74
  5.3 Characterisation of participants ............................ 77
  5.4 Problems and solutions ................................ 82
  5.5 Review ............................................... 82

x
6 Emerging Theory of Onboarding Sessions

6.1 How do experts represent the code to the newcomer? .............. 83

6.2 How do experts support newcomers in locating information? .... 85

6.3 How do experts support newcomers in making contributions? ... 85

6.4 What are the problems with onboarding sessions that reduce their value? .................................................. 86

7 Evaluation of Theory

7.1 Evaluating fit .................................................. 89

7.1.1 Characteristics of respondents ............................... 90

7.1.2 Analysis of support for statements .......................... 91

7.1.3 Summary .................................................... 93

7.2 Evaluating generalisability ....................................... 95

7.2.1 Characteristics of respondents ............................... 96

7.2.2 Experiences of onboarding ................................. 97

7.2.3 No onboarding experience ................................. 99

7.2.4 Summary .................................................... 101

7.3 Summary ....................................................... 102

8 Grounded Theory of Onboarding Sessions

8.1 How experts represent code to the newcomer: Shaping a mental model of the code ................................. 103

8.1.1 What is there - a Structural view of code .................. 104

8.1.2 How it works - an Algorithmic view of code ............... 112

8.1.3 Why it is there - a Rationale view of code ................. 115

8.1.4 What is changing - a Temporal view of code ............. 120

8.1.5 How experts represent code to newcomers: Summary .... 123

8.2 How experts support newcomers in locating information: Shaping newcomers’ information seeking strategy ........... 123

8.2.1 Establishing terminology ..................................... 124

8.2.2 Documentation .............................................. 125

8.2.3 Co-workers .................................................. 127

8.2.4 Reference materials ......................................... 129

8.2.5 Code and code artifacts ..................................... 130

8.2.6 Overall strategy .............................................. 132

8.2.7 How experts support newcomers in locating information: Summary .................................................. 132

8.3 How experts support newcomers in making contributions: Giving advice about working ............................... 133

8.3.1 General advice for efficient working .......................... 134

8.3.2 Task specific advice ......................................... 136

8.3.3 How experts support newcomers in making contributions: Summary .................................................. 138

8.4 Problems with onboarding: Session difficulties ................. 138

8.4.1 Session configuration ......................................... 139
List of Tables

2.1 Experiments relating to program comprehension theories . . 15
5.1 Characterising sessions . . . . . . . . . . . . . . . . . . . . 75
5.2 Characterising newcomers . . . . . . . . . . . . . . . . . . 79
5.3 Characterising experts . . . . . . . . . . . . . . . . . . . . 81
7.1 Results for Statement 21 . . . . . . . . . . . . . . . . . . . . 92
7.2 Results for Statement 34 . . . . . . . . . . . . . . . . . . . . 92
7.3 Support for lack of preparation . . . . . . . . . . . . . . . . 98
7.4 Support for experts forgetting code details . . . . . . . . . 99
E.1 Supported statements . . . . . . . . . . . . . . . . . . . . . . 242
E.2 Support for statement 6c . . . . . . . . . . . . . . . . . . . . 249
E.3 Support for statement 6g . . . . . . . . . . . . . . . . . . . . 249
E.4 Support for statement 8 . . . . . . . . . . . . . . . . . . . . . 249
E.5 Support for statement 9 . . . . . . . . . . . . . . . . . . . . . 250
E.6 Support for statement 11 . . . . . . . . . . . . . . . . . . . . 250
E.7 Support for statement 13 . . . . . . . . . . . . . . . . . . . . 251
E.8 Support for statement 15 . . . . . . . . . . . . . . . . . . . . 251
E.9 Support for statement 18 . . . . . . . . . . . . . . . . . . . . 252
E.10 Support for statement 21 . . . . . . . . . . . . . . . . . . . . 252
E.11 Support for statement 23 . . . . . . . . . . . . . . . . . . . . 252
E.12 Support for statement 24 . . . . . . . . . . . . . . . . . . . . 253
E.13 Support for statement 27 . . . . . . . . . . . . . . . . . . . . 254
E.14 Support for statement 28 . . . . . . . . . . . . . . . . . . . . 254
E.15 Support for statement 30 . . . . . . . . . . . . . . . . . . . . 254
E.16 Support for statement 31 . . . . . . . . . . . . . . . . . . . . 255
E.17 Support for statement 32 . . . . . . . . . . . . . . . . . . . . 255
E.18 Support for statement 33 . . . . . . . . . . . . . . . . . . . . 256
E.19 Support for statement 34 . . . . . . . . . . . . . . . . . . . . 256
E.20 Support for statement 40a . . . . . . . . . . . . . . . . . . . 257
E.21 Support for statement 40c . . . . . . . . . . . . . . . . . . . . 257
E.22 Support for statement 40d . . . . . . . . . . . . . . . . . . . . 258
E.23 Support for statement 40e . . . . . . . . . . . . . . . . . . . . 258
E.24 Support for statement 40f . . . . . . . . . . . . . . . . . . . . 258
E.25 Support for statement 40g . . . . . . . . . . . . . . . . . . . . 259
LIST OF TABLES

E.26 Support for statement 41 ........................................... 259
E.27 Support for statement 43 ........................................... 260
E.28 Support for statement 46 ........................................... 260
E.29 Support for statement 51 ........................................... 261
E.30 Support for statement 52 ........................................... 261
E.31 Support for driver recommendation ............................... 262
E.32 Support for owner recommendation ............................... 262

G.1 Codebase types ....................................................... 275
G.2 Team locations ....................................................... 275
G.3 Project types ......................................................... 276
G.4 Organisation sizes .................................................... 276
G.5 Countries .............................................................. 276
G.6 Experience ............................................................. 276
G.7 Number of projects .................................................... 277
G.8 Prevalence of onboarding sessions ................................. 277
G.9 Experience of ‘expert’ role (within ‘onboarding experience’ group) .................................................. 277
G.10 Experience of ‘newcomer’ role (within ‘onboarding experience’ group) .................................................. 277
G.11 Support for Structural view ......................................... 278
G.12 Support for Algorithmic view ....................................... 278
G.13 Support for Rationale view ......................................... 279
G.14 Support for Temporal view ......................................... 280
G.15 Support for value of Temporal information ....................... 280
G.16 Support for shaping information strategy ......................... 282
G.17 Support for advice-giving activity ................................ 282
G.18 Support for session configuration (owner role) .................. 283
G.19 Support for session configuration (director role) ................. 283
G.20 Support for session configuration (driver role) .................. 284
G.21 Support for relationship between owner and driver .......... 284
G.22 Support for session timing (relative to task) ..................... 286
G.23 Support for session timing (relative to environment setup) .... 287
G.24 Support for development environment problems after the session .................................................. 287
G.25 Support for value of environment help ............................ 288
G.26 Support for session content: during the session ................ 288
G.27 Support for session content: after the session ................. 289
G.28 Support for session content: useful activities .................... 289
G.29 Support for lack of preparation .................................... 289
G.30 Support for experts forgetting code details ....................... 290
G.31 Support for experts forgetting setup details ....................... 291
G.32 Support for project terminology issue ........................... 292
G.33 Support for value of help with terminology ..................... 292
G.34 Support for information overload issue .......................... 293

xvi
| G.35 Support for information overload effects | 293 |
## List of Figures

1. Variations of system knowledge and development skill amongst software developers. xix

1.1 A concept map for the context of this research. 2

4.1 The overall research design. 56

4.2 A fragment of transcript, coded within ATLAS.ti 64

4.3 Part of the dimensions of the concept “pointing out changing code”, recorded in ATLAS.ti 64

4.4 Building up relationships between codes in ATLAS.ti 65

4.5 Memos related to “pointing out changing code”, recorded in ATLAS.ti 66

4.6 Axial coding: grouping “pointing out changing code” and other concepts under the category of “temporal issues”, recorded in ATLAS.ti 66

8.1 Shaping a mental model of the code 105

8.2 Shaping information seeking strategy 124

8.3 Giving advice about working 133

8.4 Effectiveness of session 139

8.5 Newcomer-expert onboarding session (NEOS) theory 165
Taxonomy of Terms

The following terms have been given specific meanings in the context of this thesis:

Newcomers are skilled software developers who lack specific knowledge of the code they will work on. They are newcomers in the context of this unfamiliar system. The term is used to distinguish these developers from novices, who do not have significant development skill. Figure 1 illustrates this difference. Newcomers may lack domain knowledge, and may also struggle with unfamiliar tools as they attempt to familiarise themselves with their target system, depending on their background, but are not novices to programming in general. In other studies of onboarding, skilled developers in this position are also known as ‘apprentices’ or ‘expert apprentices’ (Berlin, 1993) and ‘new hires’ or ‘software immigrants’ (Sim and Holt, 1998). They have also been referred to as ‘novices’ (Fronza et al., 2011), but in this thesis, the terms ‘novice’ and ‘newcomer’ are not interchangeable.

Figure 1: Variations of system knowledge and development skill amongst software developers.
**Experts** are software developers who are skilled at software development in general, but who also have extensive, in-depth knowledge of the workings of a particular software system or part of a large system. They are experts in the context of this system (as opposed to newcomers, who are not familiar with this system). Experts have “two intertwined roles” (Berlin, 1993): as software developers and as information sources. Other names for experts include ‘team historians’ (LaToza et al., 2006), ‘consultants’ (Berlin and Jeffries, 1992) and ‘mentors’ or ‘veterans’ (Sim and Holt, 1998). Figure 1 illustrates their skill and knowledge.

**Onboarding** is a process by which newcomers learn to work in their team. In this thesis, the term refers to onboarding in a software development context, where the newcomer will need to learn the team’s tools, processes, culture, and the existing codebase to be maintained. (It is this code that provides the context for the ‘expert’ and ‘newcomer’ categorisations above.) In software engineering research, onboarding is also known as “naturalization of software immigrants” or “acclimation, re-tooling, start-up, ramp-up or bringing someone up to speed” (Sim and Holt, 1998). Begel and Simon (2008b) define it as “the orientation process by which new hires adjust to and become effective software developers within the corporation”. The process takes place over the newcomer’s first months or years on a project.

**Sessions** refer to (typically) one-on-one discussions between expert developers and newcomers, in which the expert provides an explanation of the code. Depending on team culture, an individual’s onboarding may include many sessions, or none at all. They are also known as ‘walkthroughs’ (Dagenais et al., 2010) and ‘consulting interactions’ (Berlin and Jeffries, 1992). Studies of software development onboarding locate sessions within a wider context of learning about a team’s tools, practices and social structure.

**Software development skill** refers to a developer’s ability to work on code in general, and includes new feature development, debugging and maintenance, and use of developer tools. Measurement of developer skill is discussed further in Section 4.4.
System knowledge is typically gained after working on a specific system for several years. This attribute is also known as ‘fluency’ within a system, defined as “the ability to complete project tasks rapidly and accurately independent of task difficulty or importance” (Zhou and Mockus, 2010). This deep understanding of the codebase, including its development history and associated tools, coupled with skill and speed at working within it, allows developers with system knowledge to guide other developers (Berlin, 1993).

The environment refers to the equipment used in day-to-day work by a developer. It includes hardware, software tools, shared networked resources and access permissions.

Push and pull are terms used in this thesis to draw a distinction between guided and unguided program exploration.

“Pulling” information refers to extracting it from unfamiliar code without guidance. Given enough time, a skilled developer can usually pull enough information to complete a given code maintenance task. This phenomenon has been studied extensively in the program comprehension literature (see Section 2.2). This activity has also been named ‘code spelunking’ (Neville-Neil, 2003), or ‘software archaeology’ in the case of legacy code (Hunt and Thomas, 2002).

“Pushing” information refers to providing information about code to a newcomer, which only a more knowledgeable developer is able to do. Studies of professional developers at work (reviewed in Section 2.6) reveal a culture of mentoring, in which expert developers pass on technical (and other) information to help new developers familiarise themselves with the code.

Push and pull are investigated further in Section 2.5.
Chapter 1

Introduction

Understanding an unfamiliar codebase is a difficult task. The majority of software developers report that understanding another’s code is a serious problem (LaToza et al., 2006). This difficulty impacts developers’ ability to maintain software (Littman et al., 1986), and is a factor in the long ramp-up time before a developer reaches full productivity on a team (Sim and Holt, 1998). In turn, these issues contribute to the industry’s well documented struggle to deliver high quality software on time.

Attempting to understand unfamiliar code is, at its core, a problem of program comprehension. Empirical studies of program comprehension (e.g. Rist, 1986; Pennington, 1987; Détienne and Soloway, 1990) have resulted in evidence for various theories. These theories have mainly focused on the information sought by programmers - i.e. the information they attempt to extract from the unfamiliar code. This work has led to the development of a program comprehension model for unguided investigations (von Mayrhauser et al., 1997), which models the developer’s variety of approaches for matching their existing, general knowledge of programming to the specific goals of the unfamiliar code. More broadly, understanding software can be considered either as an exercise in information seeking (O’Brien, 2007) or in concept and feature location (Dit et al., 2011b). Information seeking research has provided models for locating and evaluating source code (Sillito et al., 2008), documentation (Lethbridge et al., 2003) and knowledgeable colleagues (Hertzum and Pejtersen, 2000), all of which are tasks relevant to industrial software development. Equally, techniques for concept and feature location can be applied by developers seeking existing functionality in unfamiliar code as

1
part of development work. These approaches are discussed further in Chapter 2.

However, software developers joining a team are not always left to seek out information by themselves. They may also be aided by more knowledgeable colleagues, who provide information to their new team-mates with the intention of bringing them up to speed more efficiently (Berlin and Jeffries, 1992; Berlin, 1993). Compared to most studies, this scenario has a key difference. In studies of program comprehension, information seeking and concept location, the developer attempts to pull the required information from the materials available. This is typically a slow and frustrating process because the developer, working in an entirely unfamiliar landscape, is less able to judge what is important and where to find it (Sim and Holt, 1998; Dagenais et al., 2010). In contrast, a knowledgeable colleague can push this information to the newcomer and, having trodden this path before, may be in a much better position to identify and locate relevant information for their new team-mate.

The research context is illustrated in Figure 1.1.

![Figure 1.1: A concept map for the context of this research.](image)
CHAPTER 1. INTRODUCTION

1.1 Research Questions

Previous work in program comprehension, information seeking and concept location has focused on programmers pulling information from the system. However, despite the prevalence of teamwork in software development, there is a dearth of research into a complementary area of research: that of information about code that experts push to newcomers.

In this thesis, program comprehension is studied from this alternative angle. Instead of following a newcomer’s attempt at pulling information from the code and other sources, pushed explanations of code are collected from knowledgeable sources: experts, explaining the system for the benefit of newcomers, during sessions that are part of the newcomers’ onboarding process. The content of the sessions can be explored with regard to the following questions:

1. How do experts represent the code to the newcomer?
2. How do experts support newcomers in locating information?
3. How do experts support newcomers in making contributions?
4. What are the problems with onboarding sessions that reduce their value?
5. How can onboarding sessions be improved?

1.2 Scope of this study

In this study, the knowledge transfer between expert and newcomer is captured primarily through onboarding sessions, where the expert talks the newcomer through the codebase. These sessions occur as part of the onboarding process (along with other onboarding activities) and are an example of mentoring in software development.

The distinction between newcomers and novices is made to ensure that the onboarding sessions comprise system and team knowledge transfer between developers of similar skill, rather than the teaching of programming concepts to novices.

A novice software developer must first learn to program (which typically begins at university) and then learn the practices and procedures of software
development (which typically happens in their first software development position after graduating). Begel and Simon (2008a) found that in this ‘professional novice’ stage, novices learn to navigate the social structures and the development procedures, skills that are typically not well supported by university courses. They may be hindered by insecurities caused by their ‘novice’ status, preventing them from asking for help, except as a last resort.

In contrast, newcomers have made this transition and are familiar with both the general software development process and the types of knowledge needed to be effective in their role. The newcomer is making a second or subsequent move to a new project. Newcomers have the general skillset required for software development, and this, coupled with their previous experience of joining projects, distinguishes them from the professional novice. Sim and Holt (1998) offer the analogy of immigrant naturalisation to reflect the newcomer’s situation.

Experts may also be called upon to explain the use of their software to an end user. However, this study focuses on their explanation of the underlying code to another developer, and not to end users; experts may demonstrate the software to the newcomer to provide an end-user’s point of view, but this is only to provide context for the rest of the explanation.

1.3 Contributions

This research offers novel contributions to both academic understanding and industrial practice.

Firstly, the work provides a qualitative description of the current state of onboarding session practice. This contributes depth which is missing from, but contextualised by, the existing broad studies of the onboarding process in software development.

An important aspect of this description concerns the way experts explain their code and support newcomers in an unfamiliar project. This contributes a push-based view of program comprehension, information seeking and concept and feature location. This alternative view contrasts with and complements the existing literature, which is strongly biased towards the pull angle, and is particularly valuable for the evaluation of tools and theories that use expert performance as a goal - the findings illustrate that not all expert-provided information is helpful to newcomers,
so a more nuanced interpretation may be necessary.

Secondly, the work also provides a description of the problems in onboarding sessions and a set of recommendations to address them. This contributes back to industrial practice; by highlighting the effects of both the session content and the session attributes, sessions can be modified to make more efficient contributions to the onboarding process.

### 1.4 Thesis plan

Chapter 2 summarises the literature on program comprehension, information seeking, concept and feature location, and the onboarding process.

Chapter 3 reviews applicable methodologies and provides justification for the methods used in this study.

Chapter 4 describes the protocol used for the study.

Chapter 5 characterises the observations that provide the data.

Chapter 6 summarises the emerging findings from the study.

Chapter 7 describes the evaluation of the emerging theory and the changes made as a result.

Chapter 8 gives the final version of the theory after evaluation.

Chapter 9 provides a set of recommendations for practitioners and discusses their evaluation.

Chapter 10 puts the contributions in the context of the literature.

Chapter 11 describes the novel contributions of this study.
Chapter 2

Literature Review

2.1 Overview

This thesis is located at the intersection of program comprehension, information seeking, concept and feature location, and onboarding and mentoring. In this chapter, each field is reviewed for its major contributions.

Work on program comprehension has provided models for the strategies used by developers to understand unfamiliar code. These models are relevant when a newcomer is left to figure out code without help, and the models are reviewed in Section 2.2.

Models of program comprehension focus on the code as the information source. However, in practice, a developer will usually have additional information sources available, such as documentation, books and knowledgeable coworkers. Locating information sources, judging their value and locating information is the focus of information seeking. This field is reviewed in Section 2.3.

Developers are often asked to contribute code before they have a complete understanding of the codebase. In this case, the strategies and tools of concept and feature location become relevant (though the tools are rarely used in practice). Concept and feature location are covered in Section 2.4.

All three of these fields focus on unguided investigation (i.e. the ‘pulling’ of information from system artifacts). The literature on guided comprehension (or ‘pushing’ information to a newcomer) is sparse, but is
CHAPTER 2. LITERATURE REVIEW

reviewed in Section 2.5.

Finally, the literature on onboarding and mentoring is reviewed in Section 2.6. These studies take a broad view of onboarding in which understanding the code is typically just one of many activities, but this provides context for a more in-depth investigation of code knowledge pushing.

2.2 Program Comprehension

Program comprehension (or program understanding) is the process of studying unfamiliar code, resulting in increased understanding of how and why it works. The process is characterised by Boehm-Davis et al. (1996) as “reconstructing the logic, structure and goals that were used to write a computer program”.

Détienne (2002) provides a comprehensive overview of program comprehension. Many approaches derive from models of text understanding, due to its similarities: comprehension results from the combination of acquired knowledge with information derived from the text or code. An alternative to this is a problem-solving approach in which the task influences the comprehension process.

Within the text-understanding based literature, Détienne identifies three approaches. The first approach is to consider program comprehension as the application of knowledge schemas. In a programming context, these schemas correspond to plans (as described by Soloway and Ehrlich, 1984) and recognition of plans can be triggered by beacons in the code (Brooks, 1983). The second approach is one of constructing networks of relations - i.e. constructing a representation of the structure of the code. The role-oriented, hierarchical plans demonstrated by Rist (1986) are an example of this structural model. Thirdly, program comprehension can be approached as constructing a representation of a situation. An example of this situational model is Pennington’s (1987) investigation of mental models.

8
2.2.1 Beacons

Beacons, according to Brooks (1983), are sets of features in program code that indicate certain structures or operations. Brooks gave as an example a listing in which two array element values are swapped within two loops, as a ‘beacon’ for an array sort. Brooks hypothesised that programmers searched for beacons to verify their hypotheses about the code’s workings, and that discovery of beacons also prompted new and refined hypotheses.

Expert recognition of beacons was demonstrated by Wiedenbeck (1986) in two experiments. The first experiment showed that more experienced programmers recalled a short program differently from novices; they remembered key lines of the sorting procedure, while novices recalled only simple structural statements suggesting a linear reading of the code. A second experiment confirmed that experienced programmers paid close attention to these ‘beacon’ lines.

Beacons can also lead developers to infer domain-level program goals. O’Brien et al. (2004) named this behaviour ‘inference-based comprehension’ and observed heavy use of this strategy in both familiar and unfamiliar domains.

2.2.2 Plans

In programming, a ‘plan’ is a standardised structure for achieving a common goal in a program (Soloway and Ehrlich, 1984). Examples of plans include keeping a running total within a loop, searching a list for an item, and the read-analyse-report program structure. Plans are closely related to beacons in that beacons permit the recognition of plans (Détienne and Soloway, 1990). Plans have been described since the late 70s and form an important part of programming education.

Soloway and Ehrlich (1984) suggested that experts could identify plans in unfamiliar code, and hence could understand conventionally-written code more easily than novices who lacked a repertoire of plans. To test this theory, 139 students (94 novices and 45 advanced level) were asked to fill in the blanks in eight short programs. Half of the programs had a conventional structure and the other half had been modified to be ‘unplan-like’. As expected, the advanced level students performed significantly better than the novices on plan-like programs, but were reduced to near-novice level
by the unplan-like programs. Additionally, the incorrect answers to the unplan-like programs suggested that experts were incorrectly interpreting the programs as if they were plan-like.

An issue with the experiment’s design was that, without any indication of the programs’ intended purpose, participants might have arrived at one of several equally plausible interpretations, but only canonical answers were considered correct. This issue was addressed by a related experiment (Détienne and Soloway, 1990) in which the same tasks were carried out by experienced student programmers. In the latter experiment, answers other than the canonical were accepted provided they were backed up by the participant’s think-aloud comments. The results support the initial finding: experienced programmers completed the plan-like code more accurately than the unplan-like code.

In this experiment, the participants struggled to insert a missing line into one of the test programs. The authors hypothesised that this was because “the missing line is the most informative line in the plan of this program and thus, what Brooks would call a beacon”. The line in question saved the highest number seen so far in a loop. This indicates the relationship between plans and beacons: for expert programmers, recognition of a beacon allows related code to be interpreted using a plan.

Further support for expert use of plans is found in experiments on code grouping (Rist, 1986). Beginner and expert students were shown short programs (22 - 42 lines of Pascal) and asked to show which lines of code they mentally grouped together. The experts’ groupings identified plans at various levels, from global plans covering the entire program, to low-level plans expressed in only one or two lines of code. Meanwhile, the novices recognised plans in simple code, but reverted to syntactic grouping in complex code.

### 2.2.3 Mental models

The process of comprehending code involves building a mental model of it. A mental model “encodes the programmer’s current understanding of the target program. The model evolves in the course of the understanding process.” (Letovsky, 1986). The mental model described by Letovsky (1986) includes three parts: a specification of the goals of the program, the
implementation of the program, and annotations linking the two together.

Thus, comprehending a program involves building an “internal semantic structure” comprising multiple levels, from the highest level understanding of the purpose of the program, down to the lowest level details of program statements (Shneiderman and Mayer, 1979). Programmers develop the ability to generate the higher levels of this structure as their experience increases.

Pennington (1987) applied earlier theories of text comprehension to program comprehension and found evidence that the mental model has two components. The “program model”, representing the program text and its interrelationships, is built up from initial study of the program, and the “situational model”, expressing the program’s functional structure in domain terms, is built from the textbase in response to more intensive involvement with the program (e.g. modification).

The theories differ, but agree broadly that different levels of mental model are required for comprehension and that generating these levels is time and task dependent. Von Mayrhauser et al. (1997) observed two developers working on industrial maintenance tasks and found evidence for three levels in the mental model: the program model and situation model described by Pennington and a top-down model. Storey et al. (1999) aggregated these to build a set of software tool requirements for supporting mental model construction.

2.2.4 Top-down, bottom-up and the influence of domain knowledge

The approaches used by programmers can be categorised into ‘top-down’ and ‘bottom-up’ comprehension strategies. When using a top-down strategy, a programmer begins with hypotheses about the likely workings of the code, and examines the code for confirmation of hypotheses and generation of subsidiary hypotheses (for example, as described by Brooks, 1983) through recognition of beacons and plans. In contrast, bottom-up comprehension is a methodical process of understanding code line-by-line, then chunking lines together into plans to build up understanding (for example, as described by Shneiderman and Mayer, 1979; Pennington, 1987).
Top-down strategies can be further divided into expectation-based and inference-based (O’Brien et al., 2004). In expectation-based comprehension, such as that described by Brooks (1983), programmers develop hypotheses beginning from just the program’s name or purpose and seek to confirm these hypotheses in the code. An inference-based strategy, such as that described by Soloway and Ehrlich (1984), relies instead on the programmer scanning the code to identify plans and beacons that lead to hypotheses.

### 2.2.5 Integrated models of comprehension

The comprehension models discussed so far are broadly in agreement on the necessary components of comprehension: a knowledge base, a process of bringing prior knowledge to bear on the code, and a resulting mental model of the code. However, the details of these, in particular the process, vary greatly between models. Von Mayrhauser and Vans (1995a) argued that these contradictions did not indicate that any model was wrong, but that each one had only captured one of several ways in which developers could approach comprehension.


Evidence for the integrated model was provided by protocol analysis of developers working on industrial maintenance tasks. These showed developers switching strategies (and model levels) frequently during a comprehension task (e.g., von Mayrhauser and Vans, 1993, 1995b).

In practice, developers use both top-down and bottom-up comprehension. Letovsky’s (1986) think-aloud comprehension task revealed programmers asking both ‘how’ questions (indicating a known goal without matching code, i.e. a top-down process) and ‘what’ and ‘why’ questions (indicating code without a matching goal, i.e. a bottom-up process) and switching strategy opportunistically in response to cues in the code. Vessey (1985) observed programmers resorting to bottom-up ‘mental execution’ of faulty code if their initial top-down reading of
certain modules did not detect the error they were seeking. Similarly, Shaft (1995) found that most developers reported the use of metacognition to choose a strategy and were willing to switch strategy as required, for example switching to bottom-up comprehension on encountering complex calculations in code.

As suggested by Lakhotia’s (1993) introspection report, domain knowledge affects the choice of strategy: more domain knowledge permits a more top-down, hypothesis-driven approach. O’Brien et al. (2004) observed that while inference-based comprehension was used in both familiar and unfamiliar domains, the expectation-based strategy was used more in familiar domains. Bottom-up comprehension was also used in both scenarios, but in a familiar domain it became a secondary strategy used to investigate the details of code once the overall purpose was understood. This experiment extended the work of Shaft and Vessey (1995), whose coding of ‘hypothesis’ and ‘inference’ based comprehension corresponds approximately to O’Brien et al.’s (2004) ‘inference’ and ‘bottom-up’ categories, and found similar results: familiarity with the domain permits a more top-down strategy.

2.2.6 Influence of the task on comprehension strategy

Comprehension strategies and the resulting mental models vary depending on the tasks for which the knowledge will be required. This task-influenced, problem solving approach represents an alternative to the text-understanding based theories (Détienne, 2002).

The influence of the programmer’s task affects some of the comprehension experiments (and may represent a threat to validity). An example of this is the different results obtained by Pennington (1987) in which professional programmers were asked questions about a short code snippet to learn whether their mental representation corresponded to the text structure or to plan knowledge. The result was strongly in favour of a text structure representation, constructed bottom-up. However, the task given to the programmers was simply to understand and recall the 15-line program. A later experiment by Pennington (1987), using a 200-line program and a modification task, showed that the dominant mental representation changed from an initial text structure based representation.
to a more functional representation after the modification task had been completed.

### 2.2.7 Summary

The experiments described here represent almost 30 years of investigation into program comprehension. The influence of text comprehension theories can be seen in the style of the early experiments: the test materials are sets of short, abstract programs, and the tasks, intended to reveal the programmers’ behaviour, are largely unrelated to software development. Typically, there are relatively large numbers of participants, but students are often substituted for professional developers.

Over time, software development has been recognised as worthy of study in its own right, and more recent program comprehension experiments have employed professional programmers, realistic tasks and larger codebases. However, while these experiments are more applicable to professional software development, they are also more costly in terms of time and money, and hence tend to involve fewer participants.

Table 2.1 summarises the comprehension theories discussed above and the experiments providing evidence for those theories. The columns of the table summarise important aspects of the experiments that may influence their validity in real-world settings.
Table 2.1: Experiments relating to program comprehension theories

<table>
<thead>
<tr>
<th>Theory</th>
<th>References</th>
<th>Participants</th>
<th>Programs x code size (lines)</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Beacons</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wiedenbeck, 1986</td>
<td>12 novice students, 12 intermediate, 12 experienced</td>
<td>1 x 23</td>
<td>Comprehend and recall</td>
</tr>
<tr>
<td></td>
<td>Wiedenbeck, 1986</td>
<td>12 experienced programmers</td>
<td>1 x 23</td>
<td>Comprehend</td>
</tr>
<tr>
<td></td>
<td>O’Brien et al., 2004</td>
<td>8 professional programmers</td>
<td>2 x 1200</td>
<td>Familiarise, talk-aloud</td>
</tr>
<tr>
<td><strong>Plans</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Soloway and Ehrlich, 1984</td>
<td>94 novice students, 45 advanced level students</td>
<td>8 x 15</td>
<td>Cloze test</td>
</tr>
<tr>
<td></td>
<td>Soloway and Ehrlich, 1984</td>
<td>41 professional programmers</td>
<td>6 x 15</td>
<td>Recall</td>
</tr>
<tr>
<td></td>
<td>Détienne and Soloway, 1990</td>
<td>22 grad students with &gt; 2 years Pascal experience</td>
<td>8 x 15</td>
<td>Cloze test, talk-aloud</td>
</tr>
<tr>
<td></td>
<td>Rist, 1986</td>
<td>10 novice students, 7 students with &gt; 8 years programming experience</td>
<td>12 x 22-42</td>
<td>Group lines of code</td>
</tr>
<tr>
<td><strong>Mental models</strong></td>
<td>Letovsky, 1986</td>
<td>6 professional programmers</td>
<td>1 x 250</td>
<td>Modify</td>
</tr>
<tr>
<td></td>
<td>Shneiderman and Mayer, 1979</td>
<td>79 varied programmers</td>
<td>2 x 20</td>
<td>Memorise and recall</td>
</tr>
<tr>
<td></td>
<td>Pennington, 1987</td>
<td>80 professional programmers</td>
<td>8 x 15</td>
<td>Answer questions</td>
</tr>
<tr>
<td></td>
<td>Pennington, 1987</td>
<td>80 professional programmers</td>
<td>1 x 200</td>
<td>Modify and answer questions</td>
</tr>
<tr>
<td>Theory</td>
<td>References</td>
<td>Participants</td>
<td>Programs x code size (lines)</td>
<td>Task</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------</td>
<td>----------------------------------</td>
<td>-----------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Top-down and bottom-up</td>
<td>Letovsky, 1986</td>
<td>6 professional programmers</td>
<td>1 x 250</td>
<td>Modify</td>
</tr>
<tr>
<td></td>
<td>Shaft and Vessey, 1995</td>
<td>24 professional programmers</td>
<td>1 x 417, 1 x 416</td>
<td>Familiarise, talk-aloud</td>
</tr>
<tr>
<td></td>
<td>O’Brien et al., 2004</td>
<td>8 professional programmers</td>
<td>2 x 1200</td>
<td>Familiarise</td>
</tr>
<tr>
<td>Integrated model of comprehension</td>
<td>von Mayrhauser and Vans, 1995a</td>
<td>8 professional programmers</td>
<td>1 x 913</td>
<td>Answer questions</td>
</tr>
<tr>
<td></td>
<td>von Mayrhauser and Vans, 1993</td>
<td>2 professional programmers</td>
<td>1 x 40K - 500K</td>
<td>Bug fixing</td>
</tr>
<tr>
<td></td>
<td>von Mayrhauser and Vans, 1995b</td>
<td>9 professional programmers</td>
<td>1 x “large”</td>
<td>Bug fixing</td>
</tr>
<tr>
<td>Influence of the task</td>
<td>Pennington, 1987</td>
<td>80 professional programmers</td>
<td>8 x 15</td>
<td>Answer questions</td>
</tr>
<tr>
<td></td>
<td>Pennington, 1987</td>
<td>80 professional programmers</td>
<td>1 x 200</td>
<td>Modify and answer questions</td>
</tr>
</tbody>
</table>
The experiments described here all relate to the ‘pull’ scenario, and shed some light on the ways in which skilled developers approach unfamiliar code when no help is available. However, individual exploration is only one aspect of professional software comprehension. Developers also explain their code to one another, particularly during the onboarding process (Sim and Holt, 1998). As all the experiments discussed here (and all those reviewed by von Mayrhauser and Vans, 1995a) have investigated the ‘pull’ situation, there is no evidence to indicate whether this model of program understanding also holds in the ‘push’ situation, where code is actively explained by an expert to a newcomer.

2.3 Information seeking

Information seeking can be broadly defined as “the search, retrieval, recognition, and application of meaningful content” (Kingrey, 2002). These activities comprise a significant proportion of software development activity; for example, Gonçalves et al. (2011) observed Brazilian software developers at work and found that information seeking activities took up on average 32% of their time. In the software development context, information sources include code, documentation and other developers, and information is sought for a wide range of software development activities, including comprehension.

2.3.1 Models of information seeking

Information seeking behaviour can be modelled as a series of stages. Ellis and Haugan (1997) observed the information seeking behaviour of engineers and scientists performing research and identified eight stages: surveying, chaining, monitoring, browsing, distinguishing, filtering, extracting and ending. To tailor such models for software developers, O’Brien (2007) combined Ellis and Haugan’s (1997) model with three others: Wilson’s (1981) discussion of ‘information needs’, Khulthau’s model based on students using library resources, and Marchionini’s (1997) model emphasising electronic environments (which differed from more traditional environments in terms of the volume, format and accessibility of information).
The model was tested and refined using think-aloud data from two software engineers conducting a maintenance task. This resulted in a six-stage model for the information seeking process of software engineers. As in Marchionini and Wilson, the stages do not necessarily occur in order, but certain transitions between stages are more common.

**Awareness of problem** Discovering an information need. Domain and task knowledge is a prerequisite.

**Focus formulation** Generating queries about the nature and location of the problem.

**Information collection** Gathering information related to the solution. This stage has four sub-stages (integrated from Marchionini’s sub-stages): choosing a search system, formulating a query, executing the search and extracting the data.

**Examine results** Examining the relevant information in terms of the problem.

**Information Prompted Action** Feeding information back into further information-seeking activities.

**Problem solution** The information requirement has been satisfied.

Information foraging is another model for information seeking, based on biological models of predators seeking prey. Pirolli and Card (1999) adapted the model for MBA students searching for relevant documents for a company report.

Lawrance et al. (2010a) applied the information foraging theory to the software maintenance domain. In this context, the ‘predators’ are the software developers and the ‘prey’ is a bug to be fixed. The source code and documentation provide cues that give a ‘scent’ of the prey, allowing developers to home in on the defect. The programming environment (such as an IDE) corresponds to the ‘topology’ that must be navigated.

Information scent is defined as “the (imperfect) perception of the value, cost, or access path of information sources obtained from proximal cues, such as bibliographic citations, WWW links, or icons representing the sources” (Pirolli and Card, 1999). In the case of software, cues are
closely related to beacons and can also include class, method and field names (Ko et al., 2006).

The authors relate information foraging theory to earlier theories of program comprehension, in particular Brooks’s (1983) model of hypothesis-checking (discussed in Section 2.2.4). Developers were observed to make more utterances about scent than hypotheses. More detailed analysis of two data sets revealed six ‘debugging modes’: mapping, drill-down mapping, observe the failure, locate the fault, fix the fault and verify.

This model had predictive power, but, like earlier information foraging models, it assumed a fixed goal (or ‘prey’). However, during software maintenance, the goal changes rapidly as new information is discovered; the authors observed developers being ‘distracted’ by a new scent or hypothesis to investigate before fully investigating an old one. Lawrance et al. (2010b) addressed this issue with a model of reactive information foraging, in which modifications to the goal are predicted based on the information found so far. This model was able to predict the navigations of two developers more accurately than models that assumed a constant ‘prey’.

Information seeking in code is a skill that developers must learn. LaToza et al. (2007) found that experts had better information seeking strategies, allowing them to avoid wasting time on forays into unhelpful code. Although both novices and experts visited similar amounts of code, the experts’ better choices allowed them to make higher quality modifications.

Ko et al. (2006) used the information foraging model to describe the behaviour of developers working on maintenance tasks. The participants typically started by searching or browsing for relevant code, and then used file and method names to judge the relevance of the results. On finding relevant code, they foraged for related code by ‘following scents’ amongst the dependencies (for example, the class declarations, variable uses, or more implicit dependencies). Relevant code fragments are typically small and fine-grained: primarily single statements or pairs of statements, and occasionally simple methods such as accessors. They are rarely larger methods and never entire classes or interfaces. However, the source code used in the study was atypically small compared to industrial code (503 LOC across 9 classes).
2.3.2 Information sources

Major information sources for a software project include the code itself, system documentation, and other developers.

Source code

In software development, the code is often considered the most reliable source of information about the system (Singer, 1998; Seaman, 2002). Developers routinely extract information from code in order to understand existing implementations, allowing them to correct and modify programs, both in their day-to-day work (Sillito et al., 2008), and in lab-based studies (e.g. Ko et al., 2006).

To use code as an information source, developers use a range of tools from text based search to IDE navigation features; these tools are discussed further in Section 2.3.4. Extracting information from code is also a major concern of program comprehension, as described in Section 2.2.

People

The current and previous developers of the system are another important information source (Ko et al., 2007; Seaman, 2002). However, seeking information from people has its drawbacks. Problem comprehension difficulties, poor explanation skills and unhelpful attitudes impact the effectiveness of individuals as information sources, even if the information they have is valuable. (McDonald and Ackerman, 1998)

Identifying knowledgeable co-workers can be challenging. Hertzum and Pejtersen (2000) found that engineers used metainformation from relevant documents (such as authorship and distribution lists) to identify potential sources of information. Matching developers to their areas of expertise has become an important research area in its own right: see, for example, Mockus and Herbsleb (2002).

Documentation

A third important information source is documentation. Despite being created to provide information, documentation can be problematic for a
number of reasons:

The documentation’s content does not match the developer’s preferred information style. For example, Su et al. (2012) studied developers using architectural documentation and reported that some found the textual descriptions sufficient, but others cited the lack of diagrams as a hindrance.

The documentation does not exist. Creating documentation is costly, and may be abandoned when time is tight even though it is a requirement of most development methodologies. Other methodologies, notably Agile, de-emphasise comprehensive documentation in favour of working software and interactions (Beck et al., 2001). Therefore, when searching for information, a developer must consider the possibility that it was never documented.

The documentation is out of date. This is common, due to the cost of keeping it updated (Lethbridge et al., 2003). Outdated documentation can still be useful (particularly architectural overviews and inline comments) but developers must establish whether the information can be trusted.

The engineer cannot locate the documentation. In this case, developers may ask coworkers to point them towards relevant documentation (Sharif, 2012). There may be tools available for searching corporate archives of documentation, but their complexity may prove a barrier to time-constrained employees (Hertzum and Pejtersen, 2000).

There is so much documentation that useful information is difficult to locate. Dekel and Herbsleb (2009a) found that developers frequently missed important information in API documentation, even when provided with a tool (eMoose) to pull in relevant directives. The authors suggest that, in information foraging terms, the information was disguised by ‘negative scent’ - i.e. it did not appear to be worth reading until other, easier information sources had been exhausted.

Choosing an information source

A developer’s choice of information sources is influenced by the team’s configuration and culture. In a strongly colocated team, such as the one studied by Gonçalves et al. (2011), most information seeking can be conducted face-to-face rather than though email, instant messaging, or
process tools. In contrast, open source developers are rarely collocated. Email plays an important role in their information seeking, and developers use email conversations to seek information about code, people and to request documentation (Sharif et al., 2011). With fewer informal communication channels, it appears that documentation is more used and more trusted in distributed teams. However, source code is generally considered the most useful (due to its accuracy) and convenient (due to its availability) of all the information sources a developer might access, and is used more often than the rest (Seaman, 2002).

Trustworthiness is an important consideration. Hertzum (2002) noted that the accuracy of local documents can be assessed first-hand, and the qualities of colleagues as information sources are established through personal contact and by reputation. However, unfamiliar sources carry an extra cost in that their accuracy is unverified. Source code is considered highly trustworthy, although code comments may or may not be trusted (Seaman, 2002).

Software engineers often operate under tight time constraints, and this also influences their choice of information source: the accuracy of information must be balanced against the cost of locating and using it. For this reason, ease of access to the information source is an important consideration. Ease of access varies depending on the team’s location (colocated or distributed), documentation practices, tool support available, and team culture (Seaman, 2002).

### 2.3.3 Information needs

Developers’ information needs vary according to the task at hand and their familiarity with the code. Since codebases are typically large and constantly changing, even expert developers routinely seek information for change tasks. They may also require information about the task itself (for example, specific details missing from a vague bug report), and about the impact of their changes on the rest of the team and vice versa.

Information needs differ slightly depending on whether the task is enhancement or correction. For enhancement, developers need information on work history (e.g. a list of browsed locations), where to make changes to code, and the location and use of identifiers (von Mayrhauser et al.,
1997), whereas for debugging tasks, they also need domain concept descriptions and domain-program-situation model knowledge (von Mayrhauser and Vans, 1997).

Ko et al. (2007) categorised developers’ information needs into twenty-one types, ranging from highly technical (e.g. “how do I use this data structure or function”) to highly social (e.g. “what have my coworkers been doing?”). These types came from a study of seventeen software developers from the same company who were working on new development and bug fixing. Historical information was important because it allowed the developers to judge whether code was deliberately designed a particular way or whether a scenario had been overlooked. However, the question “why was the code implemented this way?” was the most difficult to answer. The authors suggest that this is one reason why “awareness” information was followed closely: by keeping up-to-date with each others’ work, developers learned the design decisions behind code as it was implemented. Developers seeking this information later were often blocked because the relevant coworker was unavailable.

Sillito et al. (2008) divided information seeking within code into four groups: finding focus points, expanding focus points, understanding a subgraph and questions over groups of subgraphs. Focus points are starting locations, depending on the task at hand (often related to feature or concept location, discussed in Section 2.4). From there, developers explored relationships between focus points and other parts of code (for example, by following inheritance hierarchies or usages). They investigated how the items in the ‘subgraph’ worked together through ‘how does it work’ and ‘why is it done that way’ type questions, but often had to revisit items because it was not possible to display all the relevant information on one screen. Finally, questions over groups of subgraphs gave developers an understanding of several subsystems together, or of a subsystem in context of the whole system.

2.3.4 Tool support for information seeking

As developers’ information needs become better understood, they can influence the requirements for development tools. Some of these are implemented in current IDEs, while others have not yet been adopted.
To locate information from code, many developers use text-based search (e.g. the ‘grep’ tool or the text search features of their editor or IDE) (Sim et al., 1998). However, converting vague hypotheses into effective search terms is difficult, and poorly chosen searches might return an unfeasibly large number of results or none at all. Starke et al. (2009) observed that developers often skimmed the results and iteratively refined their search terms to produce a manageable result set.

An alternative approach is to forage amongst related code. This is supported in current IDEs by navigation aids, allowing developers to jump between related code in different files (for example, from a variables use to its declaration, or from a method to an invocation of that method). Use of these facilities varies between populations; for example, Sillito et al. (2008) found 16 professional developers making use of a wide range of techniques for locating related code, while in Ko et al.’s (2006) lab-based study, participants mainly used the less accurate search tools.

Koskinen et al. (2004) developed a static code analysis tool named HyperSoft based on the most sought technical information, from domain concept descriptions to the nesting level of a particular procedure. Ko et al. (2006) noted that information foraging was partially supported and partially hindered by the design of the Eclipse IDE, which included some of the capabilities of HyperSoft including forward and backward call tracing. Developers spent on average 35% of their time navigating between code fragments. While some navigation is necessary, excessive switching between files (named ‘thrashing’ by de Alwis and Murphy, 2006) was found to correlate with poor quality modifications to the code (Robillard et al., 2004). The Mylar tool created by Kersten and Murphy (2005) reduced navigation time by highlighting the program elements of current interest to the developer. This Eclipse plugin maintained interest values for each element (down to the granularity of methods) that increased with element views and edits, and decayed otherwise. The authors aimed to produce a predictable encoding of programming activity, avoiding the frustrations reported by users of adaptive tools. The plugin, renamed to Mylyn, continues to be used and maintained by the Eclipse developer community.

Full or partial tool support was available for all of the 44 types of question identified by Sillito et al. (2008). However, all four groups of
questions still presented some difficulties. The authors called for improved support for more refined questions, context maintenance support, and support for integrating the results.

Code recommenders based on reactive information foraging theory have the potential to speed up the search for relevant code. Piorkowski et al. (2012) tested various recommender algorithms and found that recommenders with low momentum (i.e. using less navigation history) were more effective than those with high momentum, suggesting that goals change more rapidly in this context than in other applications of information foraging. However, the technique is reliant on developers’ ability to follow the correct scent; following a misleading scent resulted in poor recommendations.

In contrast, another type of recommender uses the navigation paths of previous work to suggest related locations to browse to. Tools such as NavTracks (Singer et al., 2005) and Team Tracks (DeLine et al., 2005) were found to be useful in some circumstances (but some tasks, notably refactoring, do not lend themselves to this type of assistance).

It is also possible to build recommender tools for documentation. One such example, the eMoose tool (Dekel and Herbsleb, 2009a) successfully drew attention to directives in the documentation of called methods. The authors interpreted this result in terms of information foraging, suggesting that the tool was boosting the ‘positive scent’ of relevant information, helping to overcome the ‘negative scent’ present in some cases.

2.3.5 Summary

Information seeking is an important skill that developers must learn in order to work effectively. However, no information source is without its drawbacks, and developers must also balance ease of access against the likelihood of finding useful information. Improving access to the right information at the right time has the potential to improve both the speed and efficiency of software development.

The study of software developers’ information seeking has yielded models and tool recommendations, many of which have been developed into tools to meet this goal. Although many were not as effective as hoped, some (most notably Mylyn) have been adopted by communities of developers.
Like program comprehension, information seeking research is (by definition) focused on the ‘pull’ model. Although other developers and their artefacts are acknowledged as information sources, the onus is on the seeker to ask for information. In the case of a newcomer, information seeking is hindered by a lack of knowledge about the problems they face and the sources that might help them.

### 2.4 Concept and feature location

Concept and feature location are techniques for matching a program’s functions to the code that implements them. These techniques address the “concept assignment problem” of program comprehension (Biggerstaff et al., 1993). Concept location allows developers to carry out maintenance without understanding the entire system (Rajlich and Wilde, 2002) which may be more practical especially in a large system (Koenemann and Robertson, 1991).

Concepts are defined by Rajlich and Wilde (2002) as “units of human knowledge that can be processed by the human mind (short-term memory) in one instance.” This definition encompasses both domain concepts and software development concepts. Most authors use “concepts” and “features” interchangeably, but others make the distinction that features are a subset of concepts. Features are defined by Eisenbarth et al. (2003) as “an observable behaviour of the system that can be triggered by the user”.

While a class may represent a concept, not all concepts are represented by a class (Rajlich and Wilde, 2002). The code for a feature or concept may be delocalised, so “location” in this context often incorporates more than one piece of code at varying levels of granularity. This property makes unaided concept location difficult, especially for a newcomer who will have difficulty judging when all the relevant parts have been found.

Concept location is considered difficult because it requires a mapping between the software development concepts or domain level concepts and the implementation level (Chen and Rajlich, 2000). Moving between these levels requires extensive knowledge, so the majority of successful concept location strategies require some input from the developer.
2.4.1 Techniques and issues

A recent survey by Dit et al. (2011b) grouped concept location techniques into different types: dynamic, static, textual, and those that combine two or more types.

Dynamic techniques

Dynamic techniques locate concepts using information from a running system. One such technique is Wilde and Scully’s (1995) “software reconnaissance”. The developer begins by identifying one set of program actions or tests that will exercise the interesting feature, and another set that will not. Code that is only executed by the first set is likely to implement the feature, providing the developer with a starting point for maintenance tasks. Improvements on this original idea include Eisenberg and De Volder’s (2005) Dynamic Feature Trace technique. This variation uses heuristics to rank code’s involvement in a feature, giving more useful results in ‘noisy’ systems. With further input from the developer, software reconnaissance techniques can also be applied to nondeterministic (e.g. distributed or multi-threaded) systems (Edwards et al., 2006).

The primary limitation of this approach is the need for suitable scenarios. Scenarios may use end-user accessible features or internal tests to exercise the code; internal tests may provide access to concepts that cannot be isolated by using the system as an end user would. However, some code (e.g. initialisation routines) is exercised in all typical uses of the system; in this case, it is difficult to create scenarios to isolate it using dynamic techniques. Creating the scenarios or tests requires some work on the part of the developer, and will be more difficult to create for a wholly unfamiliar system. The technique will fail to provide useful results in the case where the developer wrongly believes that the scenarios isolate the desired concept. Finally, dynamic techniques cannot be used on code that cannot be executed (for example, code with compilation errors or missing dependencies).

Static techniques

Static techniques analyse the structure of the code, and offer an interactive, iterative process for seeking out concept locations. The Abstract System
Dependence Graph technique (Chen and Rajlich, 2000) builds a graph in which the nodes are pieces of program code and the links are control and data flows between that code. The developer provides a starting point, allowing the system to offer related nodes by following the links. The developer then provides feedback on whether each node is relevant to the concept; relevant nodes are added to the search graph and further related nodes are offered for consideration, until the developer believes that the concept has been fully located. The Concern Graph technique (Robillard and Murphy, 2002) provides a similar process, but works with a higher level graph comprising classes, fields and methods and their calls and usages.

Static techniques are limited by the difficulties of statically identifying the effects of dynamic method binding and reflection (Robillard and Murphy, 2002). Visualisation of the process can also be an issue; without a clear representation of progress, developers report “getting lost” in the growing collection of examined and unexamined nodes (Robillard and Murphy, 2002).

Textual techniques

Textual techniques draw from information retrieval techniques, treating the code as a corpus of documents that can be processed for features. The most basic textual approach is to search the code for likely words or numbers. Sim et al. (1998) observed developers using grep and similar editor or IDE search tools to perform concept location. A striking example of this is the use of error message text taken from a bug report as a search term, in order to locate the source of the error. However, simple searching is a limited concept location technique. Without a suitable search term, a search may yield too many results to be useful, or no results at all. This is particularly difficult for newcomers who are not yet familiar with a codebase’s naming conventions or domain vocabulary (Starke et al., 2009).

More sophisticated techniques use stemming and identifier splitting to convert the code into a more tractable corpus; three such techniques were compared by Dit et al. (2011a). Information retrieval techniques, such as the vector space model (VSM) and latent semantic indexing (LSI), can be applied to non-code artifacts such as documentation to improve concept
Combining techniques

Combinations of concept location methods can overcome some of the limitations of single techniques. Dynamic concept location can be used to guide static concept location (Eisenbarth et al., 2003). Alternatively, source code rankings from dynamic and textual techniques can be combined, giving more accurate results than either technique in isolation (Poshyvanyk et al., 2007). A third approach is to apply knowledge-based filtering to the results of dynamic concept location to remove noise, for example from mouse and widget events. This approach was demonstrated on the Mozilla browser codebase (Antoniol and Guéhéneuc, 2005).

Many techniques suffer from a high rate of false positives. Dit et al. (2012) reported improved results by applying webmining techniques (HITS and PageRank) to filter irrelevant results.

2.4.2 Summary

Concept location techniques aim to assist developers with the concept assignment problem, providing tools for maintenance tasks. Regardless of the technique chosen, they are semi-automated tools, requiring input and good judgement from the developer. This in turn requires a certain level of familiarity with the software, making them difficult to use (though potentially still useful) to an unmentored newcomer. In particular, mis-ranked elements are likely to mislead tool users (Cleary and Exton, 2006).

2.5 Push and pull

Push and pull are defined by Cybenko and Brewington (1999) as: “If a user requests and receives a very specific piece of information, this is information pull. If information is sent in anticipation of the user’s need, or the agent’s response includes information not directly solicited, then the situation is characterized as information push.” This definition can be generalised to any pair of information sharing entities, including experts and newcomers.
Push and pull occur when design engineer novices consult with experts. Deken et al. (2012) categorised their interactions into expert push, expert pull, novice push, novice pull, expert delayed answer, novice delayed answer and interactive. The difference between expert push and novice pull is that the former is initiated by the expert, while the latter is initiated by the novice, but in both cases the expert is the information provider. Similarly, expert pull and novice push both feature the novice as information provider but differ by initiator. While Deken et al.’s (2012) study focused on experts and novices, this categorisation can be readily adapted for experts and newcomers.

If one dimension is the direction of information flow (push/pull) and another is the initiator (expert/newcomer), a third dimension is the level of interactivity (passive/active). Leaving information for others to find, in the form of comments or other documentation, is different from providing that same information in a mentoring session with a newcomer; similarly, actively seeking information by asking or searching is different from accidentally coming across that same information.

The studies reviewed so far largely feature newcomer pull. The program comprehension studies all feature active newcomer pull (i.e. the newcomer is tasked with understanding unfamiliar code with no help). Some, such as von Mayrhauser and Vans (1995b), use genuine industrial code and therefore also feature passive expert push, in the form of descriptive variable names and comments left by the original developers; however, in other experiments (e.g. Soloway and Ehrlich, 1984) such information was deliberately obscured to ensure that the results were not influenced by pushed information.

Information seeking studies focus on active newcomer pull. Again, information sources such as documentation are examples of passive expert push, and likewise, in conversation experts will sometimes provide more information than is requested (Berlin and Jeffries, 1992).

Concept and feature location is also primarily newcomer pull, supported by techniques and tools to reveal potentially relevant code. Recommender systems can be considered a form of push, but lack the sensitivity of an expert developer.

However, there are some studies that investigate both active and passive expert push; these are reviewed in the next section.
2.5.1 Studies of push

Murray and Lethbridge (2005b) investigated active expert push by asking expert developers to explain their code to them “as if to a newcomer” by drawing diagrams on a whiteboard. The study aimed to investigate low-level comprehension process activity on the part of the expert, rather than the transfer of information to another developer. Twenty-four developers from two companies were asked to give an informal whiteboard explanation of their system. To keep the session flowing, the authors asked a standardised set of questions about the software. The sessions were analysed for patterns of comprehension, described in more detail in Murray and Lethbridge (2005a).

Patterns include ‘baseline landmark’ (understanding a system in terms of something else), ‘necessary details’ (establishing which details are necessary for understanding and which can be ignored) and ‘temporal details’ (sequences that illustrate how the model changed over the course of the explanation). The authors recommend that these patterns be used as requirements for comprehension tools. While this study lacks ecological validity due to its artificial setup, it is among the most relevant to this thesis (although it cannot answer the research questions because no newcomers participated in the explanations).

The data from this study was interpreted in more depth by Murray (2006), resulting in ‘Snapshot theory’. A snapshot is defined as ‘an interval or moment in the development of a software model when the model is cohesive or conceptually whole’, and the expert’s explanation can be described in terms of the different categories of snapshots and sub-snapshots. Again, the author recommends that since this technique is employed in expert explanations of software, it should also be used as a basis for comprehension tools. While this is plausibly useful, no evaluation of this idea was carried out.

The author argued that this was a study of comprehension, not explanation, because the explanation exposed the expert’s mental model of the system. However, the expert was not studying an unfamiliar system for the first time, so this study is not directly comparable to earlier studies of the program comprehension process.

Some experiments unintentionally contribute an investigation of push. Lee and Kang (2012) conducted a Wizard-of-Oz study to examine the effects
of a proposed code recommender system. Although not the purpose of the research, this can be framed as ‘expert push’ in that an appropriate visualisation was pushed to the participant by a (hidden) expert as they navigated an unfamiliar system. The majority of participants found that the diagrams offered were useful guides, with one repeatedly asking how the software was able to generate such a useful diagram.

Similarly, studies of documentation and commenting can provide some insight into passive push. Dekel and Herbsleb (2009b), observing that developers often missed important points in API documentation, tested a tool for alerting them to this information. In particular, it drew attention to directives that captured “nontrivial, infrequent, and possibly unexpected information” such as the order in which methods must be called, limitations and known issues, and threading and locking considerations. The tool, ‘eMoose’, added highlights to code in the Eclipse IDE when directives were available, and made excepts available through tooltips. The 26 student participants performed significantly better on three debugging tasks (for which a directive was relevant) when eMoose was available.

Dekel and Herbsleb’s (2009b) work was extended by Monperrus et al. (2011) to build a taxonomy of API directives taken from the documentation of three large Java packages. They define a directive as “a natural-language statement that makes developers aware of constraints and guidelines related to the correct and optimal usage of an API.” The authors found that their methods for automatic directive identification would not yield high precision results; instead, the work provided information on the frequency of various types of directive and guidelines for writing effective documentation.

Information can be pushed through the code itself and the associated inline comments. Afonso et al. (2012), for example, argues for the interpretation of APIs as communication between API writers and API users, through the various artefacts that make up the API (including the code, documentation and specifications).

Developers may also leave information in code for future guidance. Chen et al. (2012) studied the use of tags (such as ‘TODO’ and ‘FIXME’) in the source code of an open-source IDE. The tags allowed in-place communication of issues, tasks, solutions, design and so on. In doing so, developers were able to communicate high-level information such as the
rationale for a design, the assumptions they made, and trade-offs that affected the implementation.

2.5.2 Summary

This section explored the various dimensions of push and pull, and reviewed the literature for theories and results. It concludes that many studies incorporate newcomer pull, and a few investigate passive expert push, but little is known about expert push in general and active expert push in particular.

This is a problem for many areas of research, because expert knowledge (while not perfect) represents the gold standard for program comprehension, information seeking, and concept and feature location. The differences in approach and content between newcomer pull and expert push highlight newcomers’ difficulties in each activity and expose the gaps that require understanding and addressing. Therefore, the lack of knowledge of expert push is problematic for the direction and evaluation of research that depends on it. In particular, some studies are instead informed by and evaluated against newcomer pull, making it difficult to judge the true impact of the proposed tools.

This research attempts to address the lack of research in this area by capturing active expert push in the context of onboarding sessions. It thus provides ecologically valid examples of expert knowledge that can be compared to the needs of newcomers to reveal the gaps that further research might address.

2.6 Onboarding

Onboarding is the process by which existing team members help a newcomer to acclimatise to working on their team. Newcomers must take in a great deal of social and technical knowledge; for software developers, this typically includes comprehension of unfamiliar code, so onboarding offers examples of program comprehension as it occurs in industry.

During onboarding, the team’s productivity suffers: not only is the newcomer unproductive, but the other team members must spend time
helping the newcomer until he or she is competent. However, onboarding is necessary to maintain team size over time. This issue has been studied in the context of many professions, including software development.

Newcomers may also be mentored by more experienced developers during onboarding. In this scenario, the expert provides the newcomer with useful information. This may include information that the newcomer would struggle to locate independently. The following studies focus on onboarding in software engineering projects. They highlight a range of issues, including the social and technical aspects of onboarding, newcomer difficulties, experts as mentors, and newcomer productivity over time.

Sim and Holt (1998) used an ‘immigration’ metaphor to represent the challenges faced by newcomers to a software project. From interviewing four newcomers to a project, they identified seven patterns in onboarding. The patterns suggested that providing newcomers with more high-level information about their new systems would decrease their reliance on the mentors. However, a pilot study of this concept had mixed results, with participants reporting that the high-level content was generally but not specifically useful.

Dagenais et al. (2010) offered a related ‘project landscape’ metaphor for the characteristics of a software project that are relevant to newcomers. Like an unfamiliar landscape, the unfamiliar project’s features can be grouped into ‘orientation aids’ and ‘obstacles’. The seventeen newcomers gradually learnt to navigate around the various IBM projects, orienting themselves through coding tasks, meetings and help from other developers. Obstacles, such as insufficient documentation and IDE installation issues, hindered the newcomer’s progress.

Berlin and Jeffries (1992) studied the ‘apprenticeship interactions’ of newcomers and experts. In this study, three HP Labs newcomers reported frustration with unfamiliar tools, embarrassment about reliance on a ‘copy and experiment’ strategy, and concern for the expert’s time that affected their willingness to interrupt and ask for help. Berlin (1993) also focused on three experts as mentors, finding that developers in this position were able to mentor without any training.

Zhou and Mockus (2010) investigated the paradox of reports that developers were fully productive after around six to twelve months (depending on project size) but could not be assigned the most critical
tasks until they had spent three years on the project. Records from ten projects confirmed that newcomer productivity plateaued at around twelve months, but interviews with 35 developers and managers revealed that development tasks varied significantly in difficulty and importance. The authors defined developer ‘fluency’ as “the ability to complete project tasks rapidly and accurately independent of task difficulty or importance”. When the task completion records were adjusted for these factors, they showed developer productivity still increasing linearly after three years, demonstrating the developers’ gradual transition from newcomer towards expert.

In follow-up work, Zhou and Mockus (2011) asked whether a newcomer’s early experiences in a project affected their chances of becoming long-term contributors. The authors defined measures for the relative sociality of a project, so that high sociality indicated more social learning and low sociality indicated more technical learning. Within the six projects studied, newcomers who joined projects with low sociality were more likely to become long-term contributors, while high project sociality was associated with more newcomers arriving. The authors hypothesise that low project sociality corresponds to more opportunities for mentoring and developing deep technical knowledge of the project, and suggest compensating newcomers who join at times of high sociality to ensure they can develop the same depth of understanding.

### 2.6.1 Newcomer issues

Moving to a new project is not easy. Despite their experience, newcomers still struggle with multiple issues: not only understanding the code, but also the programming languages (Berlin and Jeffries, 1992), asking for help (Berlin and Jeffries, 1992), worries about their own performance (Berlin and Jeffries, 1992), the toolkit (Berlin, 1993) and administrative frustration (Sim and Holt, 1998). Newcomers working remotely have extra difficulties, including social team integration, environment setup and unreliable videoconferencing facilities (Hemphill and Begel, 2011). Neville-Neil (2003) discussed some tools that a newcomer might bring to bear on a large, unfamiliar codebase, but noted later (Neville-Neil, 2008) that the tools were not keeping pace with the explosion in codebase size.
Unsurprisingly, developers who are unfamiliar with the codebase are more likely to introduce bugs when they make changes to the code (Bird et al., 2011). Newcomers, anxious about their lack of project-specific knowledge, may resort to a ‘copy and experiment’ strategy in an attempt to boost their productivity. (Berlin and Jeffries, 1992). To prevent an increase in defects, the rest of the team must spend extra time checking a newcomer’s code (LaToza et al., 2006) (Schrøter et al., 2012).

These problems are reflected in Brooks’ Law: “adding manpower to a late software project makes it later” (Brooks, 1975). Newcomers need time to ramp up to full productivity (Zhou and Mockus, 2010), and newer team members will require more help from established developers. DeMarco and Lister (1987) estimate that each newcomer will cost the team approximately three lost work-months.

Newcomers report that one of the most important factors for rapid orientation is access to the code (Dagenais et al., 2010). Early interaction with the code may be hindered by long IDE installation processes, upfront courses, unique tools to be learned and a lack of documentation. Some of these problems can be mitigated by a good mentoring culture (for example, team members can speed up a newcomer’s access to the code by helping with installation processes).

To promote early exploration, newcomers are often assigned small bug fixing tasks, chosen to introduce them to the code while minimising risks to the rest of the team; this is because maintenance is regarded as requiring less design knowledge than new feature development (LaToza et al., 2006). Newcomers to open source software projects may struggle to locate suitable bugs for their first contributions, and to discover useful information about candidate bugs. The Tesseract tool aims to assist with this scenario by enhancing search facilities with bug-specific synonym dictionaries (Wang and Sarma, 2011).

Sim and Holt (1998) observed that out of four newcomers in their study, the three who favoured a bottom-up comprehension strategy performed better at maintenance tasks. This observation may not generalise to all software developers, but does demonstrate that unguided newcomers can choose alternative strategies for learning that may impact their ramp-up time.

Berlin (1993) found that newcomers may spend a lot of time working
on a problem before consulting an expert, possibly waiting until they are completely blocked (though this can vary a great deal). This contrasts with the behaviour of experts, who consult other experts freely. Reasons for the newcomer’s reluctance include their desire to learn, their desire to make a good impression, respect for the expert’s time (Berlin and Jeffries, 1992), and a lack of clarity about the problem (Ratanotayanon and Sim, 2006). Nevertheless, a lack of documentation may drive newcomers to consult mentors.

Documentation that does not benefit most of the team may still benefit newcomers, showing that newcomers have different information needs (Cherubini et al., 2007a). As well as the design of a system, newcomers benefit from examples and rationales in the documentation (Dagenais et al., 2010).

### 2.6.2 Experts as mentors

Mentoring is believed to be an effective way for a newcomer to learn (Sim and Holt, 1998). Good mentoring culture contributes to newcomer orientation, while poor assistance exacerbates the difficulties. Newcomers report that walkthroughs are helpful for low-level technical details and the processes and tools used by the team (Dagenais et al., 2010). With expert help, newcomers can become productive team members more quickly - for example, one manager estimated that mentored developers would achieve fluency in 3 months, rather than 6-8 months when mentoring was unavailable (Zhou and Mockus, 2010).

Knowing this, the newcomer may be assigned another developer as a mentor. This is often the ‘team historian’ who has been on the team the longest, and may be the lead developer (LaToza et al., 2006) or another senior developer (Cherubini et al., 2007b; Zhou and Mockus, 2011). Typically, no training is provided for this role, but nonetheless expert programmers are able to help newcomers (Berlin and Jeffries, 1992).

Mockus (2009) investigated the effectiveness of mentoring offshore developers by examining the source code change logs of one large company. They made the reasonable proposition that “the effectiveness of expertise transfer increases over time as the organization improves its offshoring practices”. Surprisingly, this was not supported by the data, leading to
two possibilities; either the expertise transfer was already perfect, or more likely, the experts were not learning from previous offshoring attempts. The study also confirmed that knowledge transfer is easier with newer, smaller codebases, and with a smaller number of newcomers per mentor.

With their wide-ranging knowledge of the system, mentors are able to ‘jump-start’ the newcomer’s mental model of the code (LaToza et al., 2006). Amongst Microsoft teams that used visualisations, diagrams (particularly sketches) played a crucial communication role in onboarding sessions, and reverse-engineering tools could also be useful (Cherubini et al., 2007b).

Mentors are able to provide hard-to-find information; for example, experts not only recommend documentation but also rate its accuracy (Berlin and Jeffries, 1992). When a newcomer asks a question, the mentor may provide incidental information along with the answer: project history, programming strategies and tool usage (Berlin and Jeffries, 1992), design decisions and tradeoffs (Berlin, 1993). This broad education on the whole project supplies the newcomer with the background needed to maintain and extend the code with good judgement.

Conversations are wide-ranging and interactive, and both expert and newcomer can steer the topic and negotiate the level of detail (Berlin and Jeffries, 1992).

Mentors usually provide low-level technical details that are immediately useful to the newcomer’s task at hand (Sim and Holt, 1998). In order to discuss technical details, it is crucial for both developers to be able to see the code, and conversations are artifact-centred (Berlin and Jeffries, 1992). This also serves to ensure mutual understanding (Berlin, 1993).

On large systems, even the experts do not remember all the details. For this reason, it is common for experts to reconstruct their understanding of the code during mentoring (Berlin, 1993).

When mentoring a newcomer, experts must change their behaviour. Abrahamsson et al. (2011) compared patterns of tool use in pair programming by expert-expert and expert-newcomer pairs. Initially there were large differences, indicating that mentoring is a very different activity to day-to-day development. Over time, the differences fade, showing the newcomer’s gradual progression towards expert status. However, mentoring was inefficient for the expert (Sim and Holt, 1998).

Experts must also remember that newcomers have different information
needs. For example, Fritz et al. (2010) developed a degree-of-knowledge tool for source code, but the developers predicted that the results would be not useful to newcomers because APIs were rarely included.

2.6.3 Summary

Most of the studies discussed here have a broad scope, investigating the whole process of onboarding and including a wide range of social and technical issues. They paint a picture of the difficulties of onboarding, both for the newcomer and the team, and the vital role that mentoring plays in bringing newcomers up to speed efficiently. Unfortunately, they provide only a glimpse into the practices and content of mentoring in software engineering projects, opening up the field for further investigation.

While few developers have been studied, the work so far has produced a largely consistent view. The studies of experts as mentors show that they appear able to identify the parts of the system and documentation that are most useful to the newcomers, whereas newcomers themselves struggle to do so. The experts can ‘push’ this information to them (sometimes when asked, and sometimes when they perceive that the newcomer needs it). To date, there has been little investigation of the content or value of this ‘pushed’ information about software.

2.7 Discussion

To establish the background for the research questions, this chapter has reviewed program comprehension, information seeking, concept and feature location, push and pull, and onboarding. To make sense of these often disparate areas, Nakakoji et al. (2010) show how the literature can be grouped into three schools of thought. The earliest is the psychology-centred approach epitomised by the early program comprehension experiments, focusing on the programming skills of individuals. The second school is process-centred, investigating the step-by-step actions of developers on artifacts; this would include information seeking and concept and feature location. The third is
developer-centred, focusing on social processes, tools and environments. This includes the studies of onboarding and expert push, and likewise this thesis is developer-centric.

The relevant literature is almost entirely rooted in the pull scenario. These studies have resulted in theories of program comprehension and information seeking, and these theories have influenced the design of program understanding tools. However, in a pull situation, the developer’s strategy for discovering information is unlikely to be optimal because the developer is working on unfamiliar content. This is a fundamental problem with this approach, resulting in tools that enable faster navigation but do not provide real guidance. See, for example, the thrashing experienced in many IDEs and the effects of misleading cues in Piorkowski et al.’s (2012) recommender tool (Section 2.3.4).

An expert mentor, on the other hand, is in a much better position to identify and locate the most important information and push this guidance to the new developer. However, as shown by the review, investigations of the content of actively pushed information are few, and the onboarding studies that have been carried out have a wide scope that misses the detail of technical conversation. This study will begin collecting and analysing the content of onboarding sessions, to balance the existing pull-centric view of comprehension and inform strategies for understanding unfamiliar code.
Chapter 3
Methodology

Software development has its roots in computer science, but software engineering is also a human activity. As a result, software development research draws on a wide range of methodologies and techniques. This is reflected in the literature review in the previous chapter, in which methodologies range from carefully controlled laboratory experiments to longitudinal on-site case studies, and data is collected from a variety of sources including practitioners, code and tools. This chapter considers a range of techniques in order to select an appropriate methodology for the study.

3.1 Quantitative and Qualitative Research

Research methodologies can be characterised as quantitative or qualitative methods (Creswell, 2009, p.19). Quantitative methods typically employ statistical analyses to test predetermined hypotheses. The hypothesis being tested determines the structure of the measurements to be made. The measurements are based on large amounts of quantitative data, which permits an assessment of the statistical significance of the findings. Outliers in the data may be disregarded in order to build a statistical model that closely matches the majority of the data, and is therefore more likely to generalise.

An example is Soloway and Ehrlich’s (1984) research on program comprehension. The authors applied a text comprehension theory to programming to generate a hypothesis, and then designed an experiment
to test it. A large number of subjects (193) participated in the experiment and the results were statistically significant, supporting the authors’ conclusion that programmers use plans and rules of discourse to comprehend code. An attempt to investigate this using qualitative methods would not have provided this strong indication of the scale of the effect.

Another example concerns the precision-and-recall paradigm used in the evaluation of many concept location support tools (for example, Marcus et al., 2004). Using this approach, the tool’s performance is compared to a hypothetical ideal in terms of its precision (how many of the locations found are correct) and recall (how many of the correct locations are found) as percentages for selected queries. These two measures, performed on large codebases, allow tool performance to be scored and compared.

In contrast, qualitative methods operate on qualitative data (text, images and other non-numeric sources) in order to investigate complex phenomena that are difficult to quantify - in particular, human behaviour and characteristics. (Seaman, 1999). They are well suited to exploratory, descriptive studies. Good examples are provided by Berlin and Jeffries (1992) and Berlin (1993) in exploratory studies of expertise in software development. The ultimate motivation for the study was to develop tools to assist developers with consulting interactions, but rather than begin with hypotheses about tool requirements, the authors began by collecting qualitative data. Industrial programmers were provided with tape recorders and asked to record any mentoring conversations, and the recordings were contextualised through interviews. Analysis of this data yielded a description of consulting interactions and some of the issues that tool builders would need to consider. A quantitative approach, focused on measuring rather than listening, would not have allowed the participants to fully express the issues they experienced and so would not have provided a good basis for requirements gathering.

Qualitative and quantitative methods can also be combined in mixed method studies, providing context for and insight into experimental results. An example is Détiennie and Soloway’s (1990) replication of Soloway and Ehrlich’s (1984) quantitative experiment on program comprehension discussed earlier. In the original experiment, participants’ responses were labelled ‘correct’ or ‘incorrect’ in order to judge whether
the hypothesis was supported, but the replication also collected think-aloud data from the participants. This allowed for the possibility of different ‘correct’ answers when the participant gave a logically valid reason for that answer. While the quantitative results confirmed the original hypothesis about plan-like code, the qualitative data provided evidence for at least four different comprehension strategies (and a hint of a fifth) that not only explained why the original hypothesis was correct, but revealed the variations within it.

The previous chapters have established that little is known about the information exchanged in software engineering onboarding. As a result, this study is exploratory, and this is reflected in the research questions (Section 1.1). This is in contrast to well-established areas such as program comprehension, where hypotheses are developed from theories in the literature and experimentally and quantitatively tested. Not enough is known about onboarding in software engineering to create quantitative, testable hypotheses about it. Instead, the research questions are best answered through qualitative investigation and description.

### 3.2 Methodologies for exploratory studies

When researching a new area, one approach is to borrow a theory from a related area and apply it. One example, drawn from the literature review, is the application of text comprehension theories such as schema theory to program comprehension (Soloway and Ehrlich, 1984). However, this is a risky approach. It is possible that this technique will generate some insights - for example, applying schema theory to program comprehension led to the development of plan theory (Détienne and Soloway, 1990). However, there are two other possibilities: one is that the theory will not fit at all and the research will have little value, and the other is that the theory will be a partial fit. The former is clearly undesirable, but the latter brings the dangers of forcing the remaining data to fit the chosen theory and disregarding information that would suggest the existence of a better model (Glaser and Strauss, 1967).

An alternative approach is to generate a new theory from the data. Where an activity is not yet well understood, it is appropriate to begin with data collection, as illustrated by this quote from Singer et al.:
CHAPTER 3. METHODOLOGY

...we did not know exactly what it was that the [software engineers] did on a day-to-day basis. [...] Consequently, we decided to begin our study of work practices by finding out what it is that [software engineers] do when they do their work. – Singer et al., 1997

Action Research (Baskerville, 1999) is a typically qualitative methodology that explores the behaviour of humans and human systems, but with a focus on problems in organisations. In this approach, researchers involve themselves in an organisation (for example, a team of employees) and actively introduce a series of changes in order to learn about the system by observing the results. Once the researchers and clients have established the research environment (i.e. the participants’ responsibilities to each other), Action Research typically proceeds as a five-step cycle of diagnosing a problem, planning an action to address it, implementing the action, evaluating its effects, and disseminating the resulting knowledge to the organisation, the scientific community, and back into the study to inform the next cycle.

With its focus on problems in systems, Action Research would be appropriate for addressing the fourth and fifth research questions, providing evidence for changes that do (and do not) address the problems of onboarding sessions. However, it is less suited to the first three questions which aim to capture descriptions of current practice as an alternative angle to pull-based program comprehension, information seeking and concept/feature location research. Given the time required to establish a research environment, it is better suited to longitudinal studies of single organisations, which may not generalise industry-wide.

Similarly, Design Science (Peffers et al., 2007) is a problem-solution focused methodology. It differs from Action Research in that it centres on a designed artifact, rather than an introduced action, to solve organisational problems. As the organisational problem is the starting point of the research, Design Science would be appropriate for answering research question five, but the organisational problems in this area are not well characterised, so it is too early to apply this methodology. Design Science, like Action Research, is less suited to the first three research questions which do not consider problems or solutions, but instead seek to understand and describe an existing phenomenon.
Grounded Theory (GT) is a research method for systematically generating new theory from data (typically, but not necessarily, qualitative). In software engineering research, examples of Grounded Theory research include Coleman and O’Connor’s (2007) investigation into software process improvement, Dagenais et al.’s (2010) description of the onboarding process, Power’s (2002) theory of requirements documentation and Hoda’s (2011) research on self-organising agile teams. These examples illustrate the applicability of this methodology for exploring the human behaviours that comprise software engineering. The research questions of this study likewise concern the activities of software engineering practitioners, this time in the process of explaining and understanding code. Thus, Grounded Theory methodology is a viable approach.

3.3 Techniques for exploratory studies

Lethbridge et al. (2005) identify three considerations that guide the choice of appropriate data collection techniques: the type of research question, the degree of contact, and the volume of data produced. They recommend that when the research questions are broad and there is little background knowledge or comparable data (as is often the case in software engineering research), case studies are an appropriate choice.

The degree of contact refers to the level of involvement of practitioners, from direct involvement though interviews and questionnaires, to no direct involvement (instead the researcher studies work artifacts such as code and documentation). This affects both the types of data that can be collected and the cost of doing so; the authors recommend that higher levels of practitioner involvement are avoided unless it is the only way to answer the research questions.

The first three research questions concern work practices, so some contact with software practitioners is required for this; analysis of code artifacts would not reveal anything of significance about the content of onboarding sessions. Lethbridge et al. (2005) identify three techniques for capturing work practices: work diaries, shadowing/observation, participant observation, and fly-on-the-wall observation. Of these, work diaries are appropriate for capturing work practices over days or weeks, but not practical for participants to fill in during a short onboarding
session. Participant observation, requiring joining a team, or shadowing a newcomer through the onboarding process, are more suitable to longitudinal studies and might not capture enough sessions to be worth the time investment. Fly-on-the-wall observation is particularly suitable for capturing collaborative activity, and observations can be timed to capture sessions.

The remaining two questions can use the observations to identify problems and possible solutions, but will also require direct input from practitioners in order to discover omissions and other non-observable session issues. This requires a technique that is inquisitive rather than observational; Lethbridge et al. (2005) identify brainstorming/focus groups, interviews, questionnaires and conceptual modelling as inquisitive techniques.

Brainstorming and focus groups are valuable for exploratory studies, but the issues raised by a group are not necessarily the most relevant issues for individuals in their daily work. The group techniques are rejected because of the logistical difficulties of scheduling meetings with multiple busy developers. Interviews and questionnaires are valuable for investigating specific questions, and interviews in particular allow some flexibility to explore unexpected answers. Both techniques suffer from inaccuracy in reporting due to events being misremembered by participants; however, the combination with observations allows some triangulation (in particular, a comparison of what participants believe they do with what they do in practice). Conceptual modelling can be used to reveal participants’ mental models of artifacts, but without domain knowledge, a newcomer’s concept model of the system is hard to interpret and would not provide clear information about the onboarding session. Thus, interviews and questionnaires (in addition to the observations) were chosen for the final two research questions.

While questionnaire answers are somewhat amenable to automated analysis, freeform question answers and recorded interviews and observations are likely to generate large volumes of data, bringing corresponding difficulties with analysis. For a qualitative, exploratory study of this type, this is unavoidable, but software tools are available to assist with managing this data.
3.4 Methodology for this study

The previous three sections set out the methodology for the study: an exploratory, qualitative study, collecting data through questionnaires, interviews and fly-on-the-wall observation techniques and analysing it through the Grounded Theory methodology.

3.4.1 The Grounded Theory methodology

In Grounded Theory research, a process of constant comparison is used to investigate phenomena and build a theory that is grounded in the data. Such a theory is improved upon, rather than destroyed, when conflicting data is found and integrated. The methodology relies on variation in the data to expose both common categories across all cases and the variations in the properties of those categories. Without variation, constant comparison of the data would reveal very little of interest. This is in contrast to quantitative experiments, where it is important to control variables and where hard-to-measure factors such as developer skill may confound the results.

Since its inception (Glaser and Strauss, 1967), the GT methodology has split into two major versions representing the differing views of its founders. Urquhart (2001) discussed the philosophical and practical differences between their approaches. This research largely follows the methodology described by Corbin and Strauss (2008), due to their position on the value of a researcher’s own experiences (discussed in Section 3.4.1) and the accessibility of their techniques, but is modulated by the experiences and advice of other GT researchers in this field (discussed in Section 3.4.1).

Analysis techniques

In Grounded Theory research, analysis and data collection are interleaved in an iterative process, so that emerging themes from the analysis can guide the collection of further data until the analysis is complete. Important techniques include open and axial coding, constant comparison, theoretical sensitivity, saturation and memoing, each discussed below.
Early analysis takes the form of open coding. This is a process of asking questions of the data in order to break it apart into concepts. It consists of going through transcripts line-by-line, asking generative questions beginning with “what is going on here?” (Corbin and Strauss, 2008). The ability to ask such questions and choose appropriate codes is known as theoretical sensitivity, and this can be developed through exercises such as asking hypothetical questions, completing sentences, looking for commonalities and surprises, and creating visualisations (Riley, 1996). Whenever a relevant incident is found in the data, it is coded with a short descriptive label. Thus, the result of open coding is a set of concepts, named by codes, that stand for the data from which they were derived.

Each time an incident is coded, it is compared to every other incident with the same code. This constant comparison is a crucial part of the GT methodology, because it allows the variation within concepts to be discovered; these variations reveal the properties of the concepts and the dimensions of those properties (i.e. the range of ways each property can vary). This is why a varied data set is crucial for GT; the more variation in the data, the more fully the properties and dimensions can be discovered.

The properties are generated by asking “what is different about this instance compared to those already seen?”, while dimensions are discovered similarly by investigating the differences in expression of the properties of a concept. This technique builds a set of properties for each concept and a range of dimensions for each property. In this way, disparate incidents in the data are broken out and combined to create full descriptions of concepts.

Open coding and axial coding are interleaved in the analysis process, but while open coding breaks the data apart into distinct concepts, axial coding is the process of recombining related concepts into more abstract categories. Axial coding also begins with comparison, but this time concepts are compared to each other. Each new candidate concept code is compared to the existing concepts; this not only prevents the duplication of concepts, but is an opportunity to begin identifying relationships between them. Relationships between concepts may also be discovered later, triggered by individual instances in the data or through the process of constant comparison.

Relationships between concepts include:
“is-a” relationships, in which several concepts can be considered examples of a new, more abstract category.

“association”, where two or more distinct concepts frequently appear together in the data.

“contradiction”, where one concept opposes another. On closer inspection, some contradictory concepts are examples of one, more abstract category with wide variation in dimensions.

Initially, every new instance coded to a category is likely to introduce new properties and dimensions. Over time, however, the category becomes saturated - i.e. new instances can almost always be described in terms of the existing properties and their dimensions, and are not adding any new information to the category. This is a good indication that the category is well understood, and new data would not bring any new insights to the analysis. However, if all the data has been analysed and an important category has not been saturated, more data (targeted towards that category) is required. This is termed “theoretical sampling”.

As coding proceeds, incidents in the data can also trigger ideas about more abstract relationships between concepts. These ideas are important because a subset of these will eventually form the backbone of the theory. Ideas are recorded in memos, which are notes that capture ideas as they occur (and are consequently somewhat rough in nature).

The implementation of each of these techniques in this research is demonstrated in Section 4.7.

GT in software engineering research

Many software engineering researchers have successfully used the GT methodology to produce software engineering theory, and have also provided methodology experience reports, field-specific guidelines and examples that have guided this study.

Adolph et al. (2011) and Hoda et al. (2012) each provide a thorough discussion of their experiences using classical (Glaserian) GT for a study on social processes that influence software team performance. They relate the core components of the methodology to more familiar software engineering concepts - for example, Adolph et al. (2011) provided a UML
model of the process, while Hoda et al. (2012) make the helpful observation that developing codes into concepts and then into categories is a familiar technique better known to software developers as ‘abstraction’. Both provide clarifications of the GT terminology which are used in this research.

Literature review

While a literature review is a necessary part of a study, its place in a Grounded Theory study is contentious. Glaser and Strauss (1967) argued that an up-front literature review is a mistake because of the likelihood of ‘grand theory’ influencing the coding process, turning the exercise from theory generation into unintentional theory validation. More recently, Corbin and Strauss (2008) take the more pragmatic position that related literature can be used with care and can sometimes be treated as data, but that a comprehensive review typical of quantitative studies is not required.

Acknowledging the difficulty this ongoing difference presents to doctoral students, Adolph et al. (2011) suggest instead a two phase approach to the related literature. First, a “broad review positioning your research within the current knowledge landscape”. Once theory has been generated, a more detailed literature review compares and connects it with existing theory. This approach has been followed in this study; the first review is presented in Chapter 2 and the second review is in Chapter 10.

Researcher background

As GT is a technique for generating theory from data, holding preconceived ideas about the phenomena under study is potentially a problem, introducing the risk of forcing the data to fit those ideas (Glaser and Strauss, 1967). Ideally, ‘ethnographic strangeness’ (e.g. Robinson et al., 2007) permits a unique perspective on activity that might otherwise be taken for granted.

However, Corbin and Strauss (2008) take the view that preconceived ideas are unavoidable, so it is better to acknowledge the background of the researcher and allow for this influence. Since this topic is drawn from my own experiences as a software developer, this is the only option available to
me and I set out my background and experiences here in order to identify biases and preconceptions.

I studied computer science at high school and University, but most programs I created during this period were written from scratch, so there was no explanation of existing systems. However, some projects were undertaken as a group, bringing code communication issues to light. The University course did not cover onboarding issues or program comprehension techniques.

I took software development positions for two summer jobs, one University work placement and one graduate position. Thus, not including the first job where I was a novice, I have been a newcomer on three occasions, and have been introduced to around seven different commercial projects.

The four companies varied greatly in their onboarding procedures. At one end of the spectrum, I was left alone with hard copy project documentation, but no workstation, for the first week. At the other, at a company accustomed to onboarding undergraduates on placement, my team had already set up a workstation, created all the necessary accounts and identified a simple bug to be fixed. With their guidance I was able to check in code on my first day at work. At another company, I participated in their longer-term onboarding process in which I spent a few weeks shadowing different engineers involved in each of the main projects taking place. The explanation sessions offered by these engineers also varied, from long ‘braindumps’ offering an overwhelming amount of content, to narrowly focused context, specific to a bugfix in progress. Later, at the same company, I helped to onboard another newcomer, and reused code from an abandoned project with neither documentation, comments, nor any knowledgeable colleagues available. The striking difference between the best and worst onboarding procedures, and the resulting differences in productivity and frustration, inspired this research.

These experiences have given rise to some assumptions and beliefs about onboarding. Stating these beliefs helped to identify where they might impact the study:

- Both the experts and the newcomers are motivated for the newcomer to succeed (i.e. to be able to work independently).

- Experts are chosen for onboarding sessions based on expertise (but
sometimes excused due to their workload) and are not encouraged to plan an onboarding session strategy; nor given any training.

- The newcomer does not need to know everything at once (and providing more information than can be used is also a waste of the expert’s time), but ultimately they will need a broad and deep understanding of the code in order to become a key member of the team.

- Developers vary in their preferences for the media to transmit explanations: some prefer textual documentation, or diagrams, or face-to-face conversation. Thus, the same procedure is not equally natural or effective for every newcomer, expert and pair thereof. My own preference was for diagrams. The impact of this can be seen in the second pilot study (Section 4.1).

- Onboarding practices vary greatly between companies. Onboarding sessions take place in many companies but not all.

There is also value in being a ‘cultural insider’ (Coleman and O’Connor, 2007); in this case my background allowed me to interpret technical terms and in-jokes, and encouraged developers to include these details in their responses.

**Reviewing the research questions**

The GT methodology does not begin with hypotheses about the situation under study. However, it is difficult to avoid implicit hypotheses in the research questions, and it is worthwhile to examine the questions (in light of the previous section) to make these explicit. Taking the first one as an example (“During onboarding sessions, how do experts represent the code to the newcomer?”), this question assumes that onboarding sessions are conducted, that the session includes an expert and a newcomer according to the definition in this thesis, that the expert describes the code, and that the expert’s description is useful to the newcomer. The other questions can be similarly deconstructed. Some of these assumptions are reasonable given the literature and my own experiences (e.g. that sessions take place); others can be checked through data collection (e.g. the participants’ expert and
newcomer credentials) and some are important considerations (e.g. bearing in mind, during interviews and analysis, that the expert might not have a deliberate strategy for representing code to the newcomer).

Evaluation

Evaluated against the standards of more traditional empirical methods (e.g. Kitchenham et al., 2002), inductive methods such as Grounded Theory may be seen as lacking in scientific rigour. However, such research is evaluated differently to research that aims to validate an existing theory. Methodologies such as Grounded Theory provide accepted, systematic procedures for generating theory from rich data and evaluating the results appropriately (Corbin and Strauss, 2008).

Corbin and Strauss (2008) identify ten criteria for the evaluation of GT, and further criteria to judge the quality of the research. While the majority of these (such as “variation” and “sensitivity”) can be judged from the theory itself, the first criterion, “fit”, requires further input. Fit is defined by Corbin and Strauss as “Do the findings resonate/fit with the experience of both the professionals for whom the research was intended and the participants who took part in the study?”

Thus, evaluation of fit requires the input of two groups: the original participants, and a diverse group of developers who had not participated in the study until this point. Evaluating fit is then a matter of testing whether both populations agree that the findings match their own experiences.

To ensure that their theory ‘fit’ the participants’ experiences, Dagenais et al. (2010) re-interviewed seven of the original participants to check whether the theory rang true for them. Meanwhile, Hoda et al. (2012) described presenting their emerging theory to previously uninvolved practitioner groups for evaluation. This study is evaluated by both original and new participants, and this evaluation is described in more detail in Chapter 7.

3.5 Summary

Given the exploratory nature of this research and the systematic underpinnings of the approach, the Grounded Theory methodology is an
appropriate choice to provide structure to this study. Other researchers have provided experience reports and guidelines that influenced the application of the methods to these research questions. The next chapter gives the details of the resulting research methods used.
Chapter 4

Research Design

This chapter sets out how the research was performed, guided by the Grounded Theory methodology discussed in the previous chapter.

The data collection began with two pilot studies outlined in Section 4.1. Finding participants (Section 4.2) was a challenge for this study, and the approach, obstacles and workarounds are discussed here. The data collection techniques evolved with each new participant, and are outlined in Section 4.3. In particular, establishing each participant’s level of expertise was problematic and this is discussed in Section 4.4.

Each set of data required some processing before it could be analysed, and some custom software was produced to streamline the process. This is described in Section 4.5. Once prepared, each set could be incorporated into the analysis through a process of constant comparison and a hierarchy of coding. The technique is detailed in Section 4.7, and the software support is described in Section 4.6.

Once a theory has been developed, it will need evaluation with respect to fit and generalisability. The evaluation techniques are described in Section 4.8.

The overall research design is illustrated in Figure 4.1.

4.1 Pilot studies

Pilot studies allow aspects of the research design to be tested before the research begins in earnest, avoiding costly mistakes. The first pilot study was primarily conducted to test the data capture techniques. A fellow
student explained some software to me, and made a point of using the whiteboard and pen-and-paper diagrams to allow the flexibility of the recording to be tested. The session was not intended to be realistic, and was not used in the final analysis. Instead, it revealed the need for both screen recording software and a movable camcorder to capture onboarding sessions (the camcorder did not provide a readable view of the screen, but was needed to capture the conversation and the activity away from the computer). The session also revealed the need to play these videos simultaneously for transcription. Techniques for doing so are discussed in Section 4.5.

The difficulty of finding enough participants represented a risk to this study. A backup option was to act as a newcomer and have developers explain their code to me. This concept was tested in the second pilot study,
and in an attempt to add realism to the session, I agreed to make a small modification to the code afterwards.

The flaws with this approach were immediately obvious. I found it difficult to ‘switch off’ analysis of the session as it happened, leaving me unsure whether I was influencing the direction and content of the session to match my ideas, or whether second-guessing my interactions had made me an unrealistic ‘newcomer’. Additionally, having the same newcomer for all the sessions would have restricted the natural variation in software explanations. For example, I have a strong preference for diagrams (as noted in Section 3.4.1) but some developers may not find them as useful. If I always requested diagrams, this variation would not appear in the collected data. Thirdly, the expert’s motivation for explaining the system would not be as strong because the ‘newcomer’ would not be a long-term contributor to the project. Accordingly, both the second pilot study and its protocol were discarded.

### 4.2 Finding participants

Onboarding sessions often occur when a new person joins a team. However, arranging to capture these sessions was a challenging aspect of this study: not only must willing participants be found, but they must be about to begin onboarding a newcomer in the near future in order to participate.

Ideally, in a Grounded Theory study, participants are chosen for their ability to shed new light on emerging concepts in the analysis (i.e. theoretical sampling). However, due to the difficulty of recruiting participants, this study employs convenience sampling, with modifications to the collection procedures to investigate concepts as required by the analysis.

A large number of software engineers were approached and asked to participate. Personal contacts (reached through friends and colleagues of myself and my supervisors) were most often willing to participate. However, the timing of the request was important: many responses were positive but there were no onboarding sessions planned in the near future.

One participant was contacted by responding to recruitment emails sent to the University, on the basis that having recruited a developer, onboarding would be the natural next step. Along similar lines, I visited engineering careers fairs to talk to representatives of companies that employed software
developers. While the response to my proposal was frequently positive, no participants were recruited this way. Attempts to find participants through social media (on LinkedIn software engineering groups and on Twitter) also yielded no responses.

Prospective participants and their managers needed to know exactly what participation I was asking for and how long it would take. To meet this need, I developed a PDF document, giving a short overview of my research and the exact steps that I would require from participants (including setting up NDAs). I continued to evolve this information sheet in response to feedback, common questions and to reflect changes to the data collection. The final version of the sheet is included in Appendix A.

The lessons learned from this process were also presented to the Psychology of Programming Interest Group (Yates, 2012).

### 4.3 Data collection

Data collection consisted of recording the onboarding session, participant questionnaires, and followup interviews.

#### 4.3.1 Recording the onboarding session

For each set of participants, the first data collected was the recording of the onboarding session. Following the success of the first pilot study, the same observation methods were used. A camcorder pointed at the screen recorded the session from the newcomer’s point of view. The camcorder was small and mounted on a portable tripod, so that it could be moved to record off-screen activity such as diagram sketching, and its small size made it less intrusive. I sat nearby throughout the session in order to move the camcorder as required (and to check that unexpected participants, such as nearby co-workers who were asked questions, were willing to be recorded); otherwise I had no interaction with the participants during the session. The content of the screen was recorded using screen capture software. This was set up to include capture of the pointer, and allowed text and pointer actions to be seen clearly on the resulting video. The exact software used depended on the participant’s operating system.
It is well known that the presence of an observer can alter participant behaviour. Despite the efforts to make the setup unobtrusive, participants occasionally showed a sudden awareness of the recording (particularly after making a mistake or a joke) or rearranged something in the environment to improve the recorded view. One newcomer also stopped taking notes after being told by another participant that the recording would be made available after the session, raising the possibility that other participants made similar assumptions. However, there were no other indications of altered behaviour, and the participants’ assistance with the environment indicated their acceptance of the situation.

4.3.2 Questionnaires

After each session, the participants were asked to fill out an online questionnaire (using SurveyMonkey.com) about their backgrounds as software engineers. The questionnaire also asked some open-ended questions about their experiences of onboarding sessions; to avoid these questions from influencing the session, the questionnaire was presented afterwards. These questions allow some triangulation and comparison of the way people say they conduct onboarding sessions with the way they do so in practice. The initial questionnaire was tested by postgraduate students to ensure that the questions were clear, and modifications were made based on their feedback.

In the Grounded Theory methodology, analysis begins as soon as the first data is available. This allows the researcher to alter the subsequent data collection to address emerging themes in the data. Accordingly, the questions evolved as the cases and analysis progressed. For example, as familiarity with the IDE and with design patterns emerged as issues affecting the sessions, subsequent participants were asked to rate their familiarity with these tools.

The questions were also reviewed each time before participants were asked to complete them online, because the various cases revealed that apparently straightforward questions could sometimes be difficult to answer. For example, in one case a participant had worked at the company as a contractor, then worked elsewhere, and then returned as an employee; thus the answer to question “for how long have you worked here?” could not be
expressed in numeric terms without losing these important details.

Duration of employment is not, by itself, a good indicator of expertise, so further questions in the questionnaire aimed to establish each participant’s level of programming expertise. This issue is discussed in more depth in Section 4.4.

The final iteration of the questionnaire is available in Appendix B.

4.3.3 Followup interviews

After conducting two case studies, it became clear that one important aspect could not be inferred from the video of the onboarding session. The sessions themselves provided very little information on whether the newcomer found the exercise useful; that is, whether effective information transfer was taking place. To address this, from the third case study onwards, the newcomers were interviewed about the effectiveness of the session. To permit an informed response to this question, the interviews were conducted after the newcomer had attempted to modify the code under discussion (exceptions were made when it was clear that the newcomer would never modify the code). Thus, their responses were drawn from genuine experience, rather than prediction of future performance.

The interviews were semi-structured, with an evolving set of standard questions followed by questions particular to a session. To avoid biasing the participants’ answers, the standard questions were always asked first. Leading questions were avoided. The semi-structured format allowed surprising responses to be explored more deeply. The set of standard questions, as used for the final interview of the study, can be found in Appendix C.

Like the onboarding sessions, the interviews were recorded, although the interviews only required audio capture. Although Glaser and Strauss (1967) does not recommend this practice, other researchers have noted that recording reduces the chance of forgetting data, or later misinterpreting a participant’s words. In the case of interviews, it also allows the researcher to focus properly on the conversation and to follow up responses with deeper questions (Adolph et al., 2011; Hoda et al., 2012).

As predicted by Corbin and Strauss (2008, p. 28), participants often
volunteered more information after the interview had officially ended. This included further reflections on the onboarding session, demonstrations of tools, and example diagrams. I added an open question to the end of each interview for final comments. Most participants offered more detail on a previous topic which I was then able to capture.

4.4 Measuring software development skill

The experts and newcomers who provide the data for this study are distinguished from novices by their skill in software development. Thus, it is necessary to check their level of skill to ensure that the data is indeed drawn from experts and newcomers, and not from novice developers who have different needs.

For this study, it is not crucial to measure skill with a high degree of accuracy, since no quantitative results rely on this measurement. However, it is important to establish that participants could not be considered novices, and to document the measurements used so that the findings can be evaluated in light of future understanding of skill measurement.

Accordingly, three measures were used to approximate a measure of skill.

Experience (in years) Although this is not by itself a strong measure of skill (Vessey, 1985), it remains worth asking because participants with little or no previous experience are almost certain to be novices. Participants were asked about personal as well as professional project experience, since both develop skill.

Number of languages known This has been demonstrated to correlate well with skill for less experienced developers (Sheppard et al., 1979).

Self-reported expertise Participants were asked to rate their skills with various aspects of development activity (Feigenspan et al., 2012). Self-reporting is not reliable, so these answers were considered in light of the session content (for example, evidence of difficulties with the programming language would cast doubt on a self-reported language expert).

The combination of experience, languages and self-reported expertise provide sufficient background to distinguish experts and newcomers from
novices. The questions used can be found in Appendix B and the analysis is detailed in Section 5.3.

4.5 Preparing data

To prepare the data for analysis, and to develop familiarity with it, all of the onboarding sessions and interviews were transcribed as soon as possible after collection. For the audio-only interviews, this was straightforward, but the onboarding sessions typically comprised two (and on one occasion, three) simultaneous videos, and no software could be found to handle this scenario. Video combination techniques proved unsatisfactory, because the detail in each video was needed at different times (for example, the focus might alternate between code on the screen and a sketched diagram, and neither an inset or an overlay was clear enough). To achieve the required playback flexibility, I used the Matroska media file format (Matroska, 2013) to contain all the videos for each session, and VLC (VideoLAN, 2013) to play them. In this way, the videos ran simultaneously in their own windows which could be resized as appropriate.

Since no video transcription software could be found to play the Matroska format, I created a plugin for the gEdit text editor (gEdit, 2013) to provide keyboard control of VLC and insertion of timestamps into the transcription. Despite this satisfactory tool setup, the transcription was an inherently laborious process. The transcripts included details from the screencapture such as method names and mouse gestures as well as the conversation, so each hour of combined session video took around ten hours to transcribe.

4.6 Tool support for analysis

While conducting this type of analysis with pen and paper techniques is a viable option (Hoda et al., 2012), tool support is available in the form of qualitative research software. The role of the software is not to perform the analysis automatically, but to assist with organising the growing collection of materials, concepts, categories and notes.

From the various packages on offer, ATLAS.ti v6 (Muhr, 2013) was
chosen for its support for video materials. ATLAS.ti supported constant comparison by making it easy to retrieve all the incidents with the same code, regardless of the session transcript in which they originated, and to replay the corresponding slice of video. (In ATLAS.ti, the concepts and categories of Grounded Theory are both represented as ‘codes’ (Friese, 2010).) The tool also supported adding comments to concepts and categories (used to develop properties), writing memos and representing relationships between codes.

Transcripts cannot capture every nuance of individual expression, so coding from the transcripts alone carried a risk of misinterpreting the participants’ actions. Here, again, the tool supported the analysis by allowing the recording and the transcript to be viewed in synch (known in ATLAS.ti as “karaoke mode”). This feature reduced the chance of misinterpretation (particularly of humour and sarcasm).

An issue with the use of qualitative analysis software is the resultant ease of building an extensive list of concepts from only a shallow analysis of the data (Adolph et al., 2011). This occurred in an early attempt at analysis, due to a failure to employ systematic constant comparison of concepts. The resulting list was unmanageable, and was discarded. Subsequently, the constant comparison technique kept the breadth and depth of concepts in balance. This demonstrated that the tools are not a replacement for good technique, but do have a valuable supporting role.

4.7 Analysis

The analysis of the data proceeded as described in Section 3.4.1, using ATLAS.ti to assist with organisation. This section demonstrates the process, starting from an example incident in the data and tracing its contribution to the identification of a high-level category.

The analysis begins with open coding. Figure 4.2 illustrates how the transcripts were coded, using ATLAS.ti to organise the process. This fragment includes a discussion about some unused (or ‘dead’) code visible on the participants’ screen, and part of the text on the left-hand side of this figure is quoted below. This incident was coded as “pointing out changing code”.

63
Figure 4.2: A fragment of transcript, coded within ATLAS.ti

S6 expert: We’ll put [the unused code] back in under certain circumstance; like if someone’s holding shift down or something we might want to do a reload.

As analysis proceeded, similar incidents were labelled with the same code. Each one was compared with the others found up to that point, and the differences between them revealed the properties of the “pointing out changing code” concept. Figure 4.3 shows a fragment of notes about its properties and dimensions, based on a total of 77 incidents that were given this code. By keeping these records, the variations within each concept are teased apart. In order to track saturation across an increasing number of concepts, a mark was made every time a new incident did not reveal any new properties or dimensions - as the number of marks increased, so saturation became more likely.

Figure 4.3: Part of the dimensions of the concept “pointing out changing code”, recorded in ATLAS.ti

The coding process also involves comparing concepts to one another and building the relationships between them. ATLAS.ti provides facilities
to record these relationships, and produce corresponding network diagrams for concepts. Figure 4.4 illustrates the relationships between “pointing out changing code” and four other concepts. These happen to be ‘association’ relationships, but all types are supported by the software.

![Diagram](image)

Figure 4.4: Building up relationships between codes in ATLAS.ti

Any part of the analysis process may trigger more wide-ranging or in-depth ideas about the relationships between concepts and their part in more abstract categories. These ideas are recorded in memos, for which ATLAS.ti has facilities. Figure 4.5 illustrates some of the memos related to “pointing out changing code”, which included ideas about the wider issues of trying to explain a static snapshot of a changing codebase.

On occasion, these ideas concerned the apparent match (or otherwise) between the data and the literature. In order to avoid these preconceived ideas influencing the analysis, they were acknowledged and recorded in memos outside ATLAS.ti and left until the theory was related back to the literature in Chapter 10.

Constant comparison and memoing leads to the identification of axial codes, which label more abstract categories of concepts. Figure 4.6 illustrates how several concepts, including “pointing out changing code” were grouped within the category ‘temporal issues’.

It is these categories, comprising many interrelated concepts that are themselves grounded in the data, that are used to generate theory with which to answer the research questions. The ‘temporal issues’ category finds its final expression as the ‘Temporal View’ described in Section 8.1.4. The emerging theory is summarised in Chapter 6 and is then evaluated.
Figure 4.5: Memos related to “pointing out changing code”, recorded in ATLAS.ti

Figure 4.6: Axial coding: grouping “pointing out changing code” and other concepts under the category of “temporal issues”, recorded in ATLAS.ti

4.8 Evaluation

As described in Chapter 3, evaluating the quality of a Grounded Theory study is itself a qualitative process.

For this study, these evaluations were carried out through online questionnaires. The results of these questionnaires, and a discussion of other evaluation criteria, can be found in Chapter 7. Finally, the evaluation of the recommendations is discussed in Section 9.6.
4.9 Summary

The final design of this research was a combination of Corbin and Strauss’s (2008) version of the Grounded Theory methodology, experience reports and guidelines from other researchers, and changes made in response to experiences in the field. The data set is characterised in Chapter 5. The analysis was supported by commercial and custom-built software tools, and the resulting theory is laid out in Chapter 6 and evaluated in Chapter 7.
Chapter 5

Empirical Study

The materials used in this study are drawn from twelve onboarding sessions across eight different organisations, resulting in twelve sets of video, nine interviews and 24 questionnaires. They involve 23 developers and two other participants. The following sections characterise the twelve cases (Section 5.1) and the variations between them (Section 5.2), and characterise the participants (Section 5.3).

5.1 Characterisation of cases

The sessions are described in the order in which the sessions took place. To preserve anonymity, each session is assigned an identifier which is used for the remainder of the thesis. The sessions are listed in more detail in Table 5.1.

S1 In this session, the expert was the primary architect and developer of an open source data visualisation tool, written in Java at a Canadian university. The newcomers (who comprise one fairly inexperienced newcomer and one novice) had already been working on the system for several months, giving them some familiarity with the system and domain, but were being introduced to an unfamiliar area (the graph layout code) in order to add new functionality to it. They began this work immediately after the session, under the expert’s supervision.

S2 This British office products wholesaler had developed in-house inventory management software in C# and SQL. Both the expert
and newcomer were experienced professional developers, but the newcomer had started work two days previously, allowing very little time to acquire system or domain knowledge. The newcomer was required to become familiar with a report-generation system and then make a change to a stored procedure (i.e. a database query subroutine). The expert was not the original author of the software. 
During the session, the expert sought advice from a non-developer colleague who had more database and domain knowledge.

S3 This Irish company was developing a toll enforcement system in C# and SQL. The newcomer had worked on other projects in the company for three months and was expected to carry out some support and maintenance on the toll system in the near future. The expert was the primary author of the system.

S4 This Irish/US startup was developing an application metrics service in Python and JavaScript. The two developers had divided the UI and back-end development between them. The UI expert was about to take a holiday but anticipated a need for some urgent changes to the UI, so the back-end developer was being introduced to the UI code.

S5 Recorded immediately after S4 at the same company, the back-end developer then explained the back-end code to the UI developer. The expert and newcomer roles were thus reversed. However, there was no immediate expectation that the newcomer would need to work on the code.

S6 This Irish academic project on network analysis was written in Java and JavaScript and was intended for commercialisation. The expert and newcomer had developed the specification together, so both had significant domain knowledge. The expert had then developed the software and was handing it over to the newcomer for further development.

S7 This session focused on a source code analysis tool written in Java, developed in an Irish university in collaboration with several industrial partners. The expert demonstrated the use of the system to an audience including two potential new developers. The expert’s
supervisor was present and this introduced different roles to the meeting.

S8 This Irish startup was developing a web app for the management of telephone calls within large companies, using Python and JavaScript. They invited one of the developers from S5 to advise them on adding metrics collection to their telephony code. For this consulting role, the S5 developer was introduced to their codebase as a newcomer. The expert was the primary author of the code.

S9 This was a second session on the same project as S7. The meeting was more about planning future work than onboarding the newcomers.

S10 An Irish team at this global corporation were developing infrastructure code in Java and Flex for another team of developers to use. In this case, the newcomer was rejoining the company after several months’ absence, and had been tasked with adding a new charting component. In addition, another developer attended the onboarding session for a ‘refresher’ on the code. The expert was an author of a related area of infrastructure code.

S11 Recorded the day after S10 at the same company, the same newcomer was introduced to the code for a management application for data storage. This layer made use of the infrastructure code from the previous day, and was explained by two more experts who were its authors.

S12 Two newcomers were joining the project recorded in S7 and S9; one as a developer and one as an observer. The new developer had already prototyped some code changes to correct a memory issue, and this approach was discussed in the session.

E1 A session at this large corporation was excluded from the study after it was discovered that it was not an onboarding session but a code review meeting, and thus all participants were already familiar with the code under review.
5.2 Variations in sessions

A GT study relies on variation in the data. All the cases in this study are examples of software development onboarding sessions, but despite this commonality, there is much variation between them. This is a natural consequence of seeking ecological validity. Despite the use of convenience rather than theoretical sampling, the cases cover a wide variety of participant skill and background, project types, domains, company sizes and cultures, and development environments. These are characterised in this section. In a quantitative study this amount of variation would make experimental results almost impossible to compare, but in a GT study, variation increases confidence in its applicability and its insights into the research questions.

5.2.1 People present

Each session included a minimum of two participants; one newcomer and one expert. However, it was common for more people to be present. There was typically one expert, but occasionally two (or one expert developer and one domain expert) who participated. Similarly, there was typically one newcomer, but sometimes there were two or three, where several people were joining a project at the same time. Besides experts and newcomers, other participants in the sessions included another developer refreshing their knowledge (in one session), a novice (in one session), a non-developer domain expert (in one session), a supervisor (in three sessions) and academics (in one session), plus, of course, the researcher as an observer (in all but one case).

The participants varied greatly in their expertise and experience; this is discussed in Section 5.3. Most were male; one newcomer, one expert and one domain expert were female.

The expert’s working relationship to the newcomer included supervisor, team lead, peer and consultant. In most cases, the participants had worked together before the session, but in three sessions the participants had first met a few days earlier.
5.2.2 Project context

The sessions cover eight different projects, of which five were commercial and three were academic. Companies ranged in size from two-person startups to corporations with thousands of employees. All three academic projects were intended for commercialisation and one was open source. Programming languages seen in the sessions included Java, C#, SQL, Python and JavaScript, and a variety of third party libraries and frameworks. Development tools included Eclipse, Visual Studio, PyCharm and the Vim text editor. Sessions were recorded in Ireland, England, Canada and the USA.

In some cases, the expert and newcomer(s) were the only developers working on the code. In other cases, the participants were part of a larger team, which in turn was sometimes one of several teams all working on the same project. Most of the software was under active development, but one project was in a maintenance phase and was rarely changed.

5.2.3 Session setup

The purpose of most sessions was to familiarise the newcomer with the code (either in general, or for an assigned task). Some sessions also included planning future work, design of new code, development, setting up the newcomer’s machine for development, or extensive demonstrations of the system.

Most often, the session revolved around one machine belonging to the expert. In one case, both the expert and newcomer brought a laptop and worked on these in parallel. One session was conducted with no computer (just a flipchart), one used the newcomer’s laptop connected to a projector, and two sessions were conducted on a machine belonging to a meeting room which was used for remote access to the experts’ machines. Usually, the expert operated the computer (i.e. had control of the keyboard and mouse) but occasionally the newcomer would take control for some or all of the session.
5.2.4 Onboarding aspects

In some sessions, the newcomer had already been assigned a development task prior to the session. In others, the newcomer had no explicit task beyond familiarisation with the code, and in one case, a task was assigned during the session. Some sessions were recorded within a few days of the newcomer joining the project. In others, the newcomer had some experience of the project; this was because the session took place later in their onboarding process (by up to three months), or because the newcomer had worked on the design or a previous version of the software. Sometimes the newcomer had domain knowledge from working on a related project or in the same building.

Following the session, some newcomers began to work on the code immediately or within a few days, but others were delayed for up to a year by other tasks.
## Table 5.1: Characterising sessions

<table>
<thead>
<tr>
<th>Session</th>
<th>People present</th>
<th>Project type</th>
<th>Languages used</th>
<th>Time since newcomer started on project</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Expert, newcomer and novice</td>
<td>Academic, open source</td>
<td>Java</td>
<td>Three months</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>Expert, newcomer and domain expert</td>
<td>Commercial</td>
<td>C# and SQL</td>
<td>Two days</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>Expert and newcomer</td>
<td>Commercial</td>
<td>C# and SQL</td>
<td>Started that day</td>
<td>Newcomer had worked there for three months previously</td>
</tr>
<tr>
<td>S4</td>
<td>Expert and newcomer</td>
<td>Commercial</td>
<td>JavaScript</td>
<td>Started that day</td>
<td>Expert and newcomer had worked together previously</td>
</tr>
<tr>
<td>S5</td>
<td>Expert and newcomer</td>
<td>Commercial</td>
<td>Python</td>
<td>Started that day</td>
<td>Expert and newcomer had worked together previously</td>
</tr>
<tr>
<td>S6</td>
<td>Expert and newcomer</td>
<td>Academic</td>
<td>Java and JavaScript</td>
<td>Started that day</td>
<td>Newcomer had previously worked on designing the system</td>
</tr>
<tr>
<td>S7</td>
<td>Expert, newcomer, two academics, supervisor</td>
<td>Academic</td>
<td>Java</td>
<td>Seven months</td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>Expert, newcomer and developer</td>
<td>Commercial</td>
<td>Python and JavaScript</td>
<td>Started that day</td>
<td>Newcomer had a consulting role</td>
</tr>
<tr>
<td>Session</td>
<td>People present</td>
<td>Project type</td>
<td>Languages used</td>
<td>Time since newcomer started on project</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------</td>
<td>--------------</td>
<td>----------------</td>
<td>----------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>S9</td>
<td>Expert, newcomer and supervisor</td>
<td>Academic</td>
<td>Java</td>
<td>Seven months</td>
<td></td>
</tr>
<tr>
<td>S10</td>
<td>Expert, newcomer and two developers</td>
<td>Commercial</td>
<td>Java and Flex</td>
<td>One day</td>
<td>Newcomer used to work at this company</td>
</tr>
<tr>
<td>S11</td>
<td>Two experts, one newcomer</td>
<td>Commercial</td>
<td>Java and Flex</td>
<td>Two days</td>
<td>Newcomer used to work at this company</td>
</tr>
<tr>
<td>S12</td>
<td>Expert, three newcomers, supervisor</td>
<td>Academic</td>
<td>Java</td>
<td>7 months / observer / 3 weeks</td>
<td></td>
</tr>
</tbody>
</table>
5.3 Characterisation of participants

The expert and newcomer were identified primarily according to their roles in the session; the expert was explaining to the newcomer. The three reasons to characterise participants were to check that the newcomer was a newcomer (not a novice), that the expert was an expert on this codebase, and to examine the relationship between them. Their survey answers provided some insight into these characteristics.

For the purposes of this thesis, a novice is a developer joining their first project, and a newcomer is joining their second or subsequent project. This distinction was made from the participants’ questionnaire responses: for example, a newcomer in S12 had one year of commercial experience but joined the current project three weeks ago, while the newcomer in S7 had no commercial experience but reported working on four previous academic projects. Only one participant was considered a novice (in S1) and this session was included because there was also a newcomer present. The newcomers are characterised in Table 5.2.

Most of the experts were the original authors of the codebase and at the time of the session were actively involved in maintaining it. Accordingly, they had a deep knowledge of the code and its background, and were able to navigate it with ease. Exceptions were S2 where the expert was neither the author nor a regular maintainer, and S7, S9 and S12 where the expert was no longer involved in maintenance and worked at another company. The experts’ backgrounds are described in Table 5.3.

As expected, on average the newcomers had similar levels of expertise to the experts (as measured by the number of languages known and self-reported programming expertise), and a little less experience (years on commercial and other projects. Some notable exceptions were S2, where the newcomer had 26 years of commercial experience to the expert’s five years, and S5, where the newcomer had 11 years of commercial experience and the expert had six.

All the experts reported high levels of familiarity with their codebase, while the newcomers had on average significantly less (as measured by self-reporting). Unexpectedly, however, some individual newcomers reported high levels of familiarity that were not reflected in their performance in the session (even those with some prior experience of the codebase). This may have been partly a timing issue: the questionnaire was issued after
the session (in order to avoid influencing it) so newcomers may have been feeling more confident as a result. The expert in S2 also reported a high level of familiarity with the code, but was neither the original author nor a frequent maintainer of it. Overall, for newcomers and non-author experts, it seemed likely that this self-reported value was often more aspirational than accurate.
## Table 5.2: Characterising newcomers

<table>
<thead>
<tr>
<th>Session</th>
<th>Commercial experience (years)</th>
<th>Other experience (years)</th>
<th>Self-reported programming expertise (1-5)</th>
<th>Familiarity with the code (1-5)</th>
<th>Languages known</th>
<th>Task</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1, newcomer 1</td>
<td>0.3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>-</td>
<td>Given during session</td>
<td>Novice, three months into the onboarding process.</td>
</tr>
<tr>
<td>S1, newcomer 2</td>
<td>0.6</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>Given during session</td>
<td>Newcomer, three months into the onboarding process.</td>
</tr>
<tr>
<td>S2</td>
<td>26</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>Known before session</td>
<td>Newcomer</td>
</tr>
<tr>
<td>S3</td>
<td>0.25</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>No task</td>
<td>Newcomer who had worked on another project in the same company.</td>
</tr>
<tr>
<td>S4</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>Known before session</td>
<td>Newcomer in this case; expert in S5 and newcomer in S8.</td>
</tr>
<tr>
<td>S5</td>
<td>11</td>
<td>11</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>No task</td>
<td>Newcomer in this case; expert in S4.</td>
</tr>
<tr>
<td>S6</td>
<td>1.5</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>No task</td>
<td>Newcomer to the code, worked previously on system design.</td>
</tr>
<tr>
<td>S7, S9 and S12</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>Known before session</td>
<td>Newcomer, following four previous academic projects.</td>
</tr>
<tr>
<td>S8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>Known before session</td>
<td>Newcomer in this case; expert in S5 and newcomer in S4.</td>
</tr>
</tbody>
</table>
## Session Commercial experience (years) Other experience (years) Self-reported programming expertise (1-5) Familiarity with the code (1-5) Languages known Task Notes

<table>
<thead>
<tr>
<th>Session</th>
<th>Commercial experience</th>
<th>Other experience</th>
<th>Self-reported programming expertise</th>
<th>Familiarity with the code</th>
<th>Languages known</th>
<th>Task</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S10 and S11</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>Known before session</td>
<td>Newcomer</td>
</tr>
<tr>
<td>S12, newcomer 2</td>
<td>1</td>
<td>4.5</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>Started before session</td>
<td>Newcomer who saw videos of S7 and S9.</td>
</tr>
<tr>
<td>S12, newcomer 3</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>No task</td>
<td>Newcomer/observer who saw videos of S7 and S9.</td>
</tr>
<tr>
<td>Average</td>
<td>5.1</td>
<td>5.1</td>
<td>4.3</td>
<td>3.1</td>
<td>5.9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Session</td>
<td>Commercial experience (years)</td>
<td>Other experience (years)</td>
<td>Self-reported programming expertise (1-5)</td>
<td>Familiarity with the code (1-5)</td>
<td>Languages known</td>
<td>Time at this company (years)</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
<td>--------------------------</td>
<td>------------------------------------------</td>
<td>-------------------------------</td>
<td>----------------</td>
<td>-------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>S1</td>
<td>0</td>
<td>20</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>Original author.</td>
</tr>
<tr>
<td>S2</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>2.4</td>
<td>Not original author.</td>
</tr>
<tr>
<td>S3</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6.1</td>
<td>Original author.</td>
</tr>
<tr>
<td>S4</td>
<td>11</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>0.3</td>
<td>Original author (UI code).</td>
</tr>
<tr>
<td>S5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>0.3</td>
<td>Original author (backend code). Newcomer in S4 and S8.</td>
</tr>
<tr>
<td>S6</td>
<td>0.2</td>
<td>12</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>0.2</td>
<td>Original author.</td>
</tr>
<tr>
<td>S7, S9 and S12</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>Original author.</td>
</tr>
<tr>
<td>S8</td>
<td>0.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.5</td>
<td>Co-author.</td>
</tr>
<tr>
<td>S10</td>
<td>12</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>12</td>
<td>Senior software engineer.</td>
</tr>
<tr>
<td>S11, expert 1</td>
<td>12</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>12</td>
<td>Principal engineer.</td>
</tr>
<tr>
<td>S11, expert 2</td>
<td>4.5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
<td>Maintainer.</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>6.0</strong></td>
<td><strong>8.9</strong></td>
<td><strong>4.5</strong></td>
<td><strong>4.5</strong></td>
<td><strong>5.7</strong></td>
<td><strong>4</strong></td>
<td><strong>-</strong></td>
</tr>
</tbody>
</table>

Table 5.3: Characterising experts
CHAPTER 5. EMPIRICAL STUDY

5.4 Problems and solutions

Field studies raise a number of technical, ethical and logistical issues. The technical issues included the installation of software for the screen capture, the ambient noise in offices, and the size of the video files which made them difficult to copy (and on one occasion resulted in corruption beyond repair). With experience, these problems were avoided in later sessions, and the lessons learned were reported in Yates (2012).

The ethical issues described by Singer and Vinson (2000); Andrews and Pradhan (2001) discuss how, in some circumstances, release of the data to co-workers (especially superiors) could have had a negative impact on the participants. In one case, a supervisor requested this data, but this would not have been ethically sound and the data was not provided. Each participant was asked for their permission to report on them (i.e. their anonymised quotes and descriptions of their projects) in all publicly accessible publications.

Sessions S9 and S10 took place on the same day in different cities, so I attended and recorded S10 and one of my supervisors recorded S9 according to the usual procedure.

5.5 Review

The twelve sessions provided a rich data set covering a variety of genuine, naturally occurring onboarding sessions. The different data sources also allow some triangulation of phenomena: in particular, comparison of experts’ description of a typical session compared to the actual session recorded, and comparison of the session with the newcomer’s perspective from the interviews. In addition, the questionnaires allow the participants’ backgrounds to be factored into the analysis.

Transcription and analysis began as soon as the first session had been collected, in accordance with the methodology and design described in Chapter 3 and Chapter 4. The next chapter describes the results of this analysis.
Chapter 6

Emerging Theory of Onboarding Sessions

Having collected the data (Chapter 5) and analysed it using the methodology described in Chapter 4, it was possible to address the first four research questions. To recap those questions:

1. How do experts represent the code to the newcomer?
2. How do experts support newcomers in locating information?
3. How do experts support newcomers in making contributions?
4. What are the problems with onboarding sessions that reduce their value?

Each of these questions is considered in turn, resulting in a theory which must then be evaluated before being presented. This short chapter gives a summary of these findings, in order to provide the necessary background for the following chapters while avoiding repetition. In Chapter 7, the emerging theory is evaluated and amended in light of the evaluation. Finally, the full version of the theory is presented in Chapter 8, and forward references to this version are provided throughout this chapter.

6.1 How do experts represent the code to the newcomer?

The experts’ representation of the codebase can be described in terms of four views (Section 8.1). The order of the views presented here does not
correspond to the session structure; session structure varied greatly, with the exception that many sessions began with a high-level overview of the architecture.

The Structural view (Section 8.1.1) provides a high-level view of the components of the system, which is described through abstraction and the characteristics of the code. Abstract views of the code include a high-level view of the major components, use of design pattern names, and naming blocks of code. Code characteristics include simplicity and complexity, similarity and difference, the cost and size of code, and its importance overall or within the current context. Together, the structure and the characteristics allow the newcomer to begin constructing a ‘map’ of the code.

The Algorithmic view (Section 8.1.2) traces the flow of execution through selected parts of the code. The selected feature or concept is first demonstrated or described. This is followed by tracing the steps of its typical execution, in varying levels of detail. This view provides a mapping between the feature and its implementation.

The Rationale view (Section 8.1.3) covers the requirements and reasons that led to the system’s current design. Rather than provide requirements documents, experts demonstrated the system from an end-user’s point of view, and showed or sketched diagrams to illustrate non-obvious business rules. Sometimes the clients were described, to add context to the requirements. Reasons behind design choices included architectural patterns, project-specific conventions, workarounds and historical effects.

Finally, the Temporal view (Section 8.1.4) informs the newcomer about previous, current and future development work that affects the code. This includes temporary fixes, changes to code conventions and explanations for other temporal artifacts that might otherwise confuse a newcomer. The Temporal view provides a crucial extra dimension to what would otherwise be a static snapshot of the codebase.
6.2 How do experts support newcomers in locating information?

While experts supply newcomers with information during the sessions, this is not a sustainable way for the newcomer to learn everything about the code. Thus, experts also use the sessions to shape the newcomer’s strategies for locating information for themselves, supplementing the newcomer’s general strategies with project-specific guidance (Section 8.2).

An important prerequisite for this is the project’s terminology (Section 8.2.1); without this, newcomers struggle to phrase queries and interpret information.

Information sources recommended by the experts include project documentation (Section 8.2.2), co-workers (Section 8.2.3), reference materials (Section 8.2.4), and code and code artifacts (Section 8.2.5). Crucially, the expert describes not only where to find these materials, but provides opinions on their qualities as information sources. This includes the validity of documentation, the expertise (or otherwise) of co-workers including the expert themselves, and the ease of use of reference materials.

The expert also provides an overall strategy for information seeking, including guidance on when it is appropriate to ask co-workers for help (Section 8.2.6).

6.3 How do experts support newcomers in making contributions?

The support provided by the expert (Section 8.3) depends on whether the newcomer has been given a task before the session. In addition to an overview of the code and an information seeking strategy, all newcomers can be provided with general advice for working on the project (Section 8.3.1), such as interpretation of project conventions, prediction of common long-running tasks and workarounds for known issues. However, when the newcomer has a development task in mind, the expert can supply more targeted advice (Section 8.3.2): pointing out relevant and irrelevant code, recommending design changes and providing test data.
6.4 What are the problems with onboarding sessions that reduce their value?

Both experts and newcomers experience difficulties during the sessions, and the characteristics of the sessions influence their effectiveness (Section 8.4).

The session’s configuration (Section 8.4.1) can be viewed in terms of three roles, each of which affects the session:

**Driver:** has control of the keyboard and mouse (i.e. operates the computer).

**Director:** decides which files to open and what code to examine (i.e. directs the content of the session).

**Owner:** the main user of the machine outside of the session.

If the expert takes the driver role, they can create a more fluid experience, but this comes at a cost to the newcomer’s level of engagement - however, newcomer driving is sometimes a source of frustration to the expert. The director role requires knowledge of the system, so it is not initially available to the newcomer, but becomes more accessible as they gain knowledge. The owner role is strongly associated with both the driver role (i.e. the owner of a machine will usually also drive) and with the provision (or otherwise) of the newcomer’s development environment. Overall, the session configuration can be seen as a tradeoff between convenience for the expert and engagement for the newcomer.

The timing of the session is important (Section 8.4.2) because without the opportunity and motivation to work on the code soon after the session, the newcomer is likely to forget its content. When sessions take place before the newcomer has a task, they are less likely to be relevant to the newcomer’s eventual work.

Newcomers have difficulty making use of the session without a working development environment (Section 8.4.3). They also have difficulty setting up their environment without access to administration rights and licences. Since the setup is typically a rare operation, experts often find that they have forgotten the details needed by the newcomer, and the documentation (if available) is often out of date. Obtaining an up-to-date copy of the source code can also be a challenge for the newcomer.
While diagrams are not unusual in sessions, newcomers commonly request two types in particular: an abstract, high-level overview and a detailed view of the area relevant to their task (Section 8.4.4). When these are not provided, some newcomers ask experts to provide these diagrams after the session, while others use diagram generation tools. However, each diagram’s context and notation must be understood in order for it to be of any use, and too much detail can be as problematic as too little.

Experts do not maintain a complete knowledge of all the code (Section 8.4.4). This presents both the expert and newcomer with difficulties in the session. Where the expert has forgotten details, or makes mistakes in the explanation, the session is harder to follow and less valuable for the newcomer. Some experts anticipate this difficulty and review or comment the code before the session.

The project’s terminology is also initially problematic for both newcomer and expert (Section 8.4.4). Without this shared vocabulary, newcomers struggle to express questions and experts have difficulty making their information understood.

Most experts either created a plan for the session, or followed an important execution path through the code (Section 8.4.4). The newcomer’s questions sometimes threatened to derail this plan, and experts had to decide whether to delay answering or to allow tangents to their planned path through the code. Even with preparation, some experts felt under pressure to fill the session with as much information as possible, which had the potential to overwhelm newcomers.

In general, the difficulties for experts occur before or during the session - for example, when reviewing the code or when a newcomer asks difficult questions. For newcomers, difficulties occur during or after the session - for example, being slowed down by a lack of correct terminology, or forgetting information afterwards.
Chapter 7

Evaluation of Theory

The theory is evaluated against two of Corbin and Strauss’s (2008) criteria: fit and applicability. These criteria were chosen for their involvement of practitioners and emphasis on practical application of the research. This chapter describes the evaluation of fit; the evaluation of applicability is described in Section 9.7.

Fit is established by asking “Do the findings resonate/fit with the experience of both the professionals for whom the research was intended and the participants who took part in the study?” (Corbin and Strauss, 2008). The fit of the theory is evaluated in Section 7.1 and Section 7.2.

7.1 Evaluating fit

The first version of the theory (described in Chapter 6) made a set of statements about the nature of onboarding in software engineering. Many of these were based directly on observations of sessions or on statements from the participants. Others were interpreted from the events of the sessions, implied by issues or omissions, or were based on a small number of occurrences.

While the former statements were grounded in the data, the latter, more interpretive statements needed to be tested for fit with the original participants to ensure that the interpretation matched their experiences.

A list of statements was generated from the first version of the theory; this resulted in approximately 140 statements ranging from well grounded to interpretative. The well grounded statements were removed from the
list and the remaining statements were formatted into a 53-question questionnaire. Each question asked the participants to indicate on a rating scale whether a session activity was useful for the newcomer or whether they would agree or disagree with a statement. A comment box was provided for each question to allow more nuanced responses especially in the case of strong opinions or ‘neutral’ answers.

The questionnaire was tested by two software engineers who were not amongst the original participants; this ensured that the questions were clear to participants from that background and that the questionnaire could be completed in a reasonable timeframe. This resulted in minor changes to the question wording and format for clarity. The questionnaire was granted ethical approval and the question set can be found in Appendix D.

Likert items can be used to score agreement with an overall position, but this approach is not appropriate here: instead, each question is considered in isolation for its support (or otherwise) of a corresponding element in the theory. Although the rating scale responses are quantitative, this data is used only to highlight responses for which the qualitative comments require consideration.

### 7.1.1 Characteristics of respondents

Out of the original 23 developer participants, 15 completed the questionnaire (a 65% response rate). Of these, six respondents were the experts in the observed sessions, six were newcomers, two had had both roles and one was an observer.

The first five questions established some background for each respondent. Two of these asked “How recently have you explained a codebase to another developer?” and “How recently has a developer explained a codebase to you?”

The responses showed that the respondents had had ‘expert’ roles more recently than ‘newcomer’ roles. Five developers had explained a codebase within the previous week and another four within the previous month, while three developers had received an explanation within the previous month and another eight within the previous three months. Thus, despite the balance between original experts and newcomers, the respondents’ opinions about the newcomer role were generally based on
less recent experience. This discrepancy is probably due to their career paths since the original data collection: those who remained in employment with the same company would have gained more expertise and the ability to help subsequent newcomers.

7.1.2 Analysis of support for statements

Analysis of the remaining questions showed that some statements had very clear support. For example, all respondents reported that if “the expert describes the high-level architecture” the newcomer would find this either ‘useful’ or ‘very useful’. Well-supported statements like this (and well-rejected reversed statements) indicated that the corresponding element of the theory had a good fit. However, for many statements the situation was less clear-cut.

To investigate the strength of support for each statement, responses were given values from 2 for “Strongly Agree” or “Very Useful” to -2 for “Strongly Disagree” or “Very Useless”. (Values were reversed for negative statements as appropriate). This allowed the statements to be sorted by the strength of their overall support, and this sorted data set is reproduced in Appendix E.

The statements with strong support are expected responses and are not discussed further. Statements that are strongly unsupported must be examined to discover how to improve the fit of the corresponding point in the theory. Between these two extremes lie statements that had a mixed response; in these cases, the responses and comments must be examined more carefully to learn whether any changes are required. This examination is described in full in Appendix E, but two examples are described here:

Example 1: Statement 21

Statement 21 reads “If the session uses the newcomer’s machine, it will be interrupted by missing configurations or data.” Respondents were expected to agree, based on S6 in which this happened, experts’ apparent preference for their own machines, and newcomers’ reports of difficulties with environment setup. The results are given in Table 7.1.
The disagreeing respondent comments “All code and project data should come from source control system” (S3 Newcomer) and several other comments agree that the setup should be documented or standardised. However, most comments concede that this idealised situation rarely occurs in practice and, that being the case, such interruptions are helpful to the newcomer because they uncover blocking problems while the expert is on-hand to assist. Thus, the statement is well-supported, and the comments also provide more insight into the session configuration issue. No changes to the theory were needed as a result of this response.

Example 2: Statement 34

Statement 34 reads “From the expert’s point of view, the ideal session does not waste time on environment setup.” Respondents were expected to agree with this, based on the apparent reluctance of most experts to include this in the sessions. The results are given in Table 7.2.

The results and comments show some support for an ‘ideal’ case in which the newcomer follows up-to-date setup instructions without help from the expert. However, many comments recognise that in practice, this documentation is not available. In this case, time that the expert spends on the newcomer’s environment setup is not considered a waste since it is necessary for the newcomer to become valuable (and, more tellingly, to prevent repeated interruptions from the newcomer). It is also an opportunity for developers to improve the setup documentation for subsequent newcomers.

The comments do not match the realities of the observed sessions,

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.1: Results for Statement 21

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.2: Results for Statement 34

92
suggesting that although experts recognise the newcomers’ difficulties with setup, they can be reluctant to help. Responses to Statement 28 (Appendix E.2) shed light on the reasons for this reluctance: experts often do not remember the details of the setup.

The theory was changed to note that experts’ reluctance to help with environment setup is not because they consider it a waste of their time. Instead, either the expert believes (usually mistakenly) that the setup instructions are adequate, or they anticipate difficulties with providing assistance because they have forgotten the details.

7.1.3 Summary

Based on this feedback from the original participants, the following changes were made to the theory to improve the fit:

**Shaping the newcomer’s mental model**

Going through any code line-by-line with a newcomer can be useful but should be considered a last resort (Tables E.2 and E.3). Similarly, stepping through code using the debugger is only useful to the newcomer for complex or delocalised code (Tables E.13 and E.16).

Pointing out dead code is useful only if the expert is giving the reasons for keeping it in the codebase, and not simply stating that the code is unused (Table E.9).

As experienced developers, newcomers are able to compare their local codebase with that under version control. However, while version control systems make the problem manageable, they do not completely eradicate temporal issues caused by concurrent maintenance or poor version control discipline. Thus, the problem is not that newcomers cannot tell if their code is up-to-date relative to the content under version control, but that they cannot tell whether the content under version control is up to date relative to the team’s activities (Table E.25).
CHAPTER 7. EVALUATION OF THEORY

Supporting information seeking

The newcomer will need to know about project-specific workarounds, environment setup details and how to run the software as an end-user. In some projects, this information is available in project documentation and the expert can refer the newcomer there. Otherwise, the expert will have to provide this information (Tables E.21, E.22 and E.24).

Supporting newcomers’ contributions

Information about off-limits code is useful to the newcomer, but should be accompanied by reasons for the limitations - for example, whether a limitation is due to permanent code ownership conventions or current priorities (Table E.28).

Problems with onboarding

Experts are more likely to resort to back-seat driving if they are themselves unfamiliar with the code and need to explore it on-the-fly. It is not the lack of thorough low-level detail that causes this behaviour in experts, but the lack of a high-level overview of the system (Tables E.7 and E.12).

Guessing answers to the newcomer’s questions is acceptable if it is clear that it is a guess, but it is preferable to provide the newcomer with guidance to locate the correct answer (Table E.27).

If the newcomer cannot find the code without a back-seat driver, it may indicate that they do not yet have a high-level grasp of the system or terminology (Table E.6).

While experts may be frustrated by session problems, a ‘smooth’ session is not as useful to the newcomer as one that runs into and then addresses their problems and misunderstandings. Environment setup in particular may feel like a ‘problem’ in the context of an explanation session, but is an important prerequisite for effective explanation (Table E.11).
Some of the expert’s choices about the session are not done to save time for the expert’s short term benefit, but for some other reason. For example, the expert’s reluctance to help with environment setup is not because they consider it a waste of their time (Table E.19). Instead, either the expert believes (usually mistakenly) that the setup instructions are adequate (Table E.21), or they have difficulty providing assistance because they have forgotten the details (Table E.14).

### 7.2 Evaluating generalisability

The previous section showed that with some minor changes, the theory is a good fit with the experiences of the original participants. The next step is to check that the resulting theory is also a good fit with the general population of software engineers - that is, whether it generalises.

To test this, a questionnaire was created with four sections. The first section (just one question) ensured that respondents had not previously been involved in the study. This is important because the theory had already been tested with that group; further responses from them would not answer questions about generalisation.

The second section asked for (non-identifying) information about their experience and workplace. This information made it possible to examine variation in respondents with respect to company size, nationality, project type and experience. This is necessary to demonstrate that while the identities of the respondents were unknown, they were drawn from a wide range of software development backgrounds. This increases confidence in the ability of this group to evaluate generalisation of the theory.

This research has so far concentrated on projects in which onboarding sessions take place, but this is not necessarily true of all projects. The third section used questionnaire logic features to separate respondents into those whose current projects featured onboarding sessions and those whose did not. The latter were asked some open-ended questions about how newcomers were brought up to speed and whether they saw any value in onboarding sessions.

Finally, developers who had experience of onboarding sessions were asked a series of questions to find out whether their experiences matched
the theory. As in the previous survey, most questions took the form of rating scales for agreement, followed by a comment box. Respondents were encouraged to provide comments, as this would allow for greater understanding of the responses especially in the case of disagreements with the theory.

The questionnaire was tested by a software engineer who had not previously been involved in the research, and some questions were modified for clarity as a result. The final question set can be found in Appendix F.

Potential respondents were contacted through personal contacts in the industry and local technology interest groups. The questionnaire was also advertised using social media (Twitter). Attempts to recruit participants for the original study in this way had yielded no responses, but for the online questionnaire this channel worked well and this accounts for much of the variation in nationality in the responses.

The survey was completed by 33 individuals. Of these, two had already participated in the research and so their responses were discarded, leaving 31 responses from previously uninvolved participants. Their responses are detailed in Appendix G.

7.2.1 Characteristics of respondents

The second set of questions established the variation in the respondents’ organisations and experience levels. The data set includes open and closed source projects (Table G.1), colocated and distributed teams (Table G.2), commercial and academic projects (mostly commercial, Table G.3), a wide range of organisation sizes (Table G.4), and locations (mostly the UK, Table G.5).

The participants reported a wide range of years of experience (Table G.6), but all participants had at least two years’ experience, so no responses were discarded on ‘novice’ grounds. To confirm this, respondents were also asked how many software projects they had worked on. All respondents reported a minimum of two projects, and many of the more experienced respondents gave answers such as “more than 30” or “too many to count”; this was a strong indication that they were no longer novices (Table G.7).
Finally, respondents were asked whether onboarding sessions take place at their current (or most recent) workplace. Seventeen reported that this happened ‘always’, ‘often’ or ‘sometimes’, while fourteen reported that this ‘rarely’ or ‘never’ happened (Table G.8). These two groups were examined in more detail to discover how their experiences compare to the theory.

These answers show that the questionnaire respondents are a varied group of software engineers, similar in variation to the original participant group. The main differences are that the ‘generalisation’ respondents are mainly from commercial projects, are more experienced on average, and some work on projects where onboarding sessions are rare or never occur.

### 7.2.2 Experiences of onboarding

Seventeen respondents reported a culture of onboarding sessions at their current (or most recent) organisations. The next two questions ensured that respondents had personal experience of onboarding sessions as either an expert (Table G.9) or a newcomer (Table G.10). Their responses also provide some insight into the frequency of their sessions. All seventeen respondents with current onboarding experience had personal experience of both roles, and thus were asked the next set of questions, to see if their experiences matched the theory.

Notably, most respondents report experience of expert and newcomer session roles in approximately the same time period (e.g. both within the last week, or both over a year ago). Future investigations might reveal whether this is due to individual preferences (some developers enjoy collaborative working more than others), company culture, or some other factor. Responses from the original participants do not show a similar correlation between answers; this suggests that individuals in that group had a stronger bias towards either expert or newcomer status, possibly due to the selection method.

Although this study focused on colocated onboarding sessions, one member of this group was part of a distributed team, showing that colocation is not a requirement for onboarding sessions. Future work might investigate the prevalence and characteristics of this variation.

Similarly to the previous evaluation stage, the responses from this group varied from clear support, through mixed response, to clear
disagreement, and in the latter two cases, the comments shed light on the reasons for disagreements. In Appendix G.2, the responses are examined for their support (or otherwise) of each aspect of the emerging theory. An example, focusing on the issue of expert session preparation, is reproduced here:

**Example: Expert knowledge and session preparation**

In the theory, some experts prepare for sessions by reviewing or commenting the code to be explained, or by planning the session’s content. However, many do not, and this is associated with making mistakes in the explanation.

The responses (Table 7.3) support the theory that most experts do not prepare for sessions. The most common reason for not preparing was the belief that, as authors of the code, they already had a comprehensive understanding of it.

**Respondent, in questionnaire:** No, I like to think I know my software.

**Table 7.3: Support for lack of preparation**

<table>
<thead>
<tr>
<th>Before explaining code to a newcomer, do you do any preparation?</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>7</td>
</tr>
<tr>
<td>No answer</td>
<td>5</td>
</tr>
<tr>
<td>Yes (reviewing code)</td>
<td>3</td>
</tr>
<tr>
<td>Yes (other activity)</td>
<td>2</td>
</tr>
</tbody>
</table>

Like the original participants, experts who reviewed the code did so because they anticipated that their knowledge could be out of date, due to recent development work or their own forgetfulness.

**Respondent, in questionnaire:** I check the latest version of the code and make sure I understand the most recent changes.

**Respondent, in questionnaire:** Maybe reopen the code to remind me what the heck it is.
This highlights the effect of temporal issues on the expert, and this link is added to the theory.

The theory also reflects the issue of experts having forgotten the details of the code they are explaining. This is somewhat supported (Table 7.4). Respondents report that forgetting code details is common in large codebases but not smaller ones:

Respondent, in questionnaire: [Agree, because:] The codebase is massive; we are bound to forget how some parts work until we look at it again.

Respondent, in questionnaire: [Disagree, because] Codebase is quite small.

Table 7.4: Support for experts forgetting code details

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

However, in the observed sessions, code details are forgotten in codebases of all sizes. One possible explanation is that code is forgotten when a developer has not interacted with it for some time, and this scenario is more common with large codebases but can also occur when a small codebase is no longer being maintained. It is also possible that many experts do not expect this to be an issue (as evidenced by the lack of preparation as well as their survey responses) but in practice it is common (as observed in sessions.) However, investigation of this issue remains for future work. While the cause is unclear, there is some support for the issue of experts forgetting code.

7.2.3 No onboarding experience

Just under half of the respondents reported that onboarding sessions rarely or never occur in their current workplaces. This raises the possibility that the theory would not generalise to all developers. A full analysis of this
CHAPTER 7. EVALUATION OF THEORY

group is outside the scope of this study, but in this section, the responses from this group are briefly examined to understand where the theory does and does not apply.

A common reason for not providing a walkthrough is that it is not feasible because the codebase is too large. This implies an all-or-nothing understanding of the term, and several respondents in this no-onboarding group reported that newcomers are offered a high-level overview or a walkthrough of relevant sections. This demonstrates that onboarding sessions, as defined in this thesis, are more common than this questionnaire initially suggested (and therefore the theory is more generalisable), but that practitioners lack a widely recognised term for this practice.

Respondent, in questionnaire: [Walkthroughs never happen:] A newcomer usually gets a general architectural overview, but at a very high level (with all questions answered on demand).

Where no onboarding session is provided, a common approach is to assign the newcomer a task and offer help on request. This learning-by-doing approach is generally considered sufficient, and is also efficient with the experts’ time.

Respondent, in questionnaire: Help is provided on request for tricky bits, but the newcomer gets familiar by fixing small bugs, not through talking.

Many respondents also report a reliance on documentation, but those writing from a newcomer’s point of view report the usual problems with documentation.

Respondent, in questionnaire: I am currently the newcomer, (developer) docs are a little thin. Code itself is good, but there are a few unexplained warts. I am now documenting projects in a wiki for subsequent developers. This is probably the second or third job where I’ve done this.

Asked whether they saw any benefit to code walkthroughs, respondents gave a wide variety of answers, illustrating the perceived pros and cons of in-person explanation.
Responder, in questionnaire: Yes. Able to offer explanations and answer questions and provide additional background (and foreground) info.

Responder, in questionnaire: No. The documentation was written for a reason, for a member of the team to take time out and explain the same information to the newcomer would be a complete waste of resources.

Responder, in questionnaire: I’ve had that once - it lasted for hours and I couldn’t remember a word.

Responder, in questionnaire: Perhaps an hour of someone’s time to go through the innards and control flow of [a feature] would have made me productive more quickly. In this instance, an hour of someone’s time wasn’t a reasonable thing to request.

These responses echo themes in the theory; in particular, the tendency to optimise onboarding for the expert (by not taking up their time) rather than the newcomer, the value of expert knowledge, and the issue of information overload.

7.2.4 Summary

The responses from this group of developers indicate that the theory is widely applicable beyond the original group of participants. However, as before, the evaluation indicated aspects of the theory that required alteration in order to improve its fit with all participants in the study. These alterations, together with those from the previous section, resulted in the final version of the theory presented in Chapter 8.

No changes are required to the first three sections of the theory (i.e. “Shaping the newcomer’s mental model”, “Supporting information seeking” and “Supporting newcomers’ contributions”). The following changes are made to the fourth section, “Problems with onboarding”:

- When the newcomer’s task is discussed within the session, this has a positive influence on the session’s relevance. This requires specific knowledge on the part of the expert (Appendices G.2.3 and G.2.4).
• The owner role can be taken by either the newcomer or the expert (Appendix G.2.4).

• The link between a lack of a task and the newcomer being diverted after the session is not as strong as it appeared from the observations (Appendix G.2.4).

• An incomplete development environment is a problem for the newcomer after the session but not necessarily during the session (Appendix G.2.4).

• A lack of expert knowledge is a problem for providing the Rationale view, because this view frequently requires understanding beyond that which can be obtained from the codebase (Appendix G.2.1).

• Temporal issues impact expert knowledge, and are one of the reasons for expert session preparation (Appendix G.2.4).

• There is little support for experts forgetting how to set up the environment. This is not considered to be a common issue (Appendix G.2.4).

7.3 Summary

The evaluation indicates that the theory has good fit (according to both the original participants in the study and a wider group of practitioners), providing a high degree of confidence in its findings. The evaluation stage also indicated the need for some minor revisions to improve the fit. The resulting theory is presented in full in the next chapter.
Chapter 8

Grounded Theory of Onboarding Sessions

The theory, refined through the evaluation described in Chapter 7, is described here from the perspective of the first four research questions. To recap those questions:

1. How do experts represent the code to the newcomer?

2. How do experts support newcomers in locating information?

3. How do experts support newcomers in making contributions?

4. What are the problems with onboarding sessions that reduce their value?

In this chapter, each question is considered in turn and the corresponding theory is reported in each section. Finally, the theory is drawn together and summarised in Section 8.5.

8.1 How experts represent code to the newcomer: Shaping a mental model of the code

The first research question concerns how experts represent the code to the newcomer. This is a core activity of an explanation session, and the sessions offered many examples of the representation of code. To help the newcomers
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

build up a model of the codebase, the experts described the code from four different viewpoints:

**Structural view** “What is there” - a high to mid level look at the components of the system. (Section 8.1.1)

**Algorithmic view** “How it works” - tracing the flow of execution through selected parts of the code. (Section 8.1.2)

**Rationale view** “Why it is there” - describing the requirements and design principles that led to the current design. (Section 8.1.3)

**Temporal view** “What is changing” - how previous, current and future development work is impacting the code. (Section 8.1.4)

The ordering of the groups presented here is arbitrary; in practice, explanation from all four viewpoints was intertwined within the sessions. The four views, and the activities that contributed to them, are illustrated in Figure 8.1. Although the sessions also touched on important resources other than the codebase, they are discussed elsewhere: this mental model is solely a model of the code.

### 8.1.1 What is there - a Structural view of code

Experts describe the current version of the code to the newcomers using two devices. The first is the use of **abstraction**. This ranges from the highest level view of the architecture of the code, through design patterns and down to naming small pieces of code. At each level, this provides the newcomer with a way to group and refer to related code, and reduces their need to examine all the details.

The second is the transmission of the **characteristics of code**. These characteristics range from the relatively factual (such as the amount of code making up a feature or the speed of an operation) to the expert’s opinions on the code’s complexity and quality.

**Abstraction of structure**

Experts provide an abstract view of the structure of code. Most important to the newcomers, and described early on in most sessions, is the high-level
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

Figure 8.1: Shaping a mental model of the code

level architecture, providing the newcomer with a rough outline of the ‘shape’ of the codebase. Typically, this comprises the major components of the code (the projects of a C# solution, the packages of a Java project, and so on). In a distributed system it includes the relationships between the components.
S6 newcomer, in interview: So he ran though the structure of the packages he used, and he showed me the layout of the project, and a rough idea of how it worked.

S3 expert: (C# projects listed in Solution Explorer in IDE) So if you collapse down all the projects, I’ll give you an overview of all the projects and how they work together.
S3 newcomer: (collapses all projects)
S3 expert: They’re all the projects involved in the whole thing. [This project] is the web application, that’s actually what the solution’s all about, and [that project] is in there because it does some of the logging stuff that we need access to. The common stuff, you’re familiar with that, that’s the common library stuff...

Experts occasionally provide diagrams of this information, but more often simply point to components in the IDE. The latter quote also demonstrates that this initial explanation takes place at a high level of abstraction with little detail (in this case, deliberately hidden from view through an IDE feature). Newcomers value abstraction; diagrams with too much detail are not helpful at this early stage because the details obscure the structure.

If high level architectural information is missing from the session, newcomers will seek it out, either from the expert or from the codebase using automated tools:

S12 newcomer: You have the [project] in two packages, the [first package] and the [second package]... can you explain just what each thing is?

S12 newcomer, in interview: [Documentation autogeneration tool Doxygen] allows you to see how packages [or] the code actually lays out rather than how it’s logically laid out based on the package structure, so it gives you a better insight into how the code is working.

Newcomers value the high-level view of the code because it improves their ability to find code, allowing them to predict where code is and is not likely to be.
S6 newcomer, *in interview*: The best way it helped me was that I could find [the relevant code] quickly, I knew where to go to look for it... when I was looking for something and I knew it was in there somewhere, I had a rough idea after him describing the structure, the layout of it, where to go.

Following this highly abstract, high level view of the code, the use of abstraction continues throughout the session. In code that uses architectural patterns such as model-view-controller, the expert identifies components according to this structure.

S4 expert: This would be a controller in the MVC pattern of things.

S10 newcomer, *in interview*: The patterns that we were using, the MVC, which is crucial. The patterns give you the structure to your code and when you’re actually adding the new features you have to follow that pattern.

At lower levels of the architecture, the expert may explain code in terms of **design patterns**, which, with one name, express the existence and relationships of a group of classes (Gamma et al., 1995). Design pattern names such as ‘observer’ (or ‘listener’), ‘command’ and ‘visitor’ have become industrial standards.

S4 expert: [This object] creates itself as a listenable, so it can be listened to for things like this series had been added or removed, the title has changed, the timeseries has changed...

There is generally an assumption that newcomers will understand these patterns and names. Newcomers were asked to rate their familiarity with design patterns in the background questionnaire (*Appendix B*); of the eight newcomers who responded, all reported at least a little familiarity with design patterns and a majority (five) reported strong familiarity (although no newcomer rated themselves as an expert). However, the sessions showed that less experienced developers may not be familiar with all the patterns in use in a system. In S1 the expert took down a reference book to teach the Command pattern to the relatively inexperienced newcomer and the novice, and in S12 the newcomer used online resources to learn the Visitor pattern.
S1 expert: So undo-redo itself uses the Command pattern.

Experts may also name code in accordance with common programming idioms. These names can be grouped into categories:

**Internal function:** commonly-occurring idioms in software, such as ‘initialisation’, ‘validation’ and ‘logging’.

S3 expert: (selects block of code) That’s just some validation there.

**Importance:** important code is often named using visceral metaphors: ‘meat’, ‘guts’ and ‘heart’.

S4 expert: The actual guts of the method are (selects method call) select a host, find a host that they’ve selected, make sure it gets pushed in the model...

**Template:** names indicating that code follows a template, such as ‘boilerplate’, ‘getter’ and ‘setter’.

S3 expert: (selects SQL code) This is just boilerplate code for creating a table.

**Transmission of the characteristics of code**

In addition to the structure of the code, experts provide information about its characteristics. These include their opinions about its attributes: its complex and simple areas, similarities and differences, bottlenecks and issues, and the core parts. Other characteristics, such as costs and code sizes, are more factual, while the importance (or otherwise) of code is largely contextual.

Developers can rank code on a scale from simple to complex, and code at either extreme is pointed out. Experts point out simplicity more often than they point out complexity. Complex code may include the use of recursion, the implementation of complex business rules, and code that is not broken up into smaller chunks. Code may be called simple if it is highly cohesive, short in length, or if it is a data structure with no other
functionality. Knowledge of simple and complex areas allows a developer to predict areas of code that are likely to be easy or difficult to work with.

S1 expert: [This class is] just really simple, there’s nothing to it, it’s an ID and a hashtable.

S3 expert: [This] is the core logic for mailmerging all the different types of documents, it’s really complex, really messy, but it does the job.

Knowledge of **similarities** and **differences** in the code can ease the newcomer’s onboarding. In particular, knowledge of similarities or commonly used idioms can change the perception of code or other artifacts from complex to simple. Design patterns and names are examples of this phenomenon, but not all idioms in a codebase have names by which developers can identify them, so instead the expert brings them to the newcomer’s awareness by pointing out their similarities and differences.

S2 expert: Then we’ve got (scrolls down to method) the PRS, that’s the same thing.
S2 newcomer: Yep.

The expert points out the **cost of code**. This provides the newcomer with some understanding of where the main bottlenecks are in terms of improving performance. Costs include processing power, execution speed, file size and development time, and may be considered in absolute or relative terms.

S6 expert: (points at goDeeper method call) goDeeper is where we go (waves hands) a level out, so goDeeper is the expensive branch.

S12 expert: A single compilation file is not gonna be huge even if you’ve a huge monster Java file, compared to the whole project it’s still gonna be small.

S5 expert: Pulling down that tiny table every ten seconds is nothing.
S8 expert: It means that you’re getting twenty thousand records rather than the thirty that have changed since the last day.

Pointing out a snippet of code implicitly communicates its importance, but this can also be communicated explicitly. Conversely, experts also recommend areas of code that can be ignored. Code is considered important if it implements a critical function of the system, and can often be ignored if it implements error handling, logging or other functions not contributing to the system’s core purpose.

S6 newcomer: So this is the method that builds the graph?
S6 expert: Yep, this is the most important method.

S3 expert: I’ll go through that in more detail, but just take note, actioner host is a critical page.

S11 expert: Here I pull in the [ID], keeping it locally. [Then this is] just all general initialisation and setup stuff. Here’s an example of a query now...

S3 newcomer, in interview: [The most important component is] the case progressor, I suppose as a functioning part of the system would be highly important because it basically keeps the whole thing flowing, it’s the heart of the system, really.

Experts often pause to point out important data structures and ID variables that carry the data through the code.

S9 expert: Eclipse has a Java model, so all your project structure is represented as a set of classes called Java elements. (points to IJAVALEMENT in diagram) Even a folder is a Java element. If you have a package in your workspace, there is a class in Eclipse representing this package.

S4 expert: If we’re looking at a specific graph (points to graph.id variable) that’s what we’re looking at, if we’re looking at a specific dashboard there’s a (points to dashboard.id variable) dashboard ID.
Importance may also be contextual (e.g. code for producing GUI styling, which itself is not a crucial function) or relative (more important than some other part of the code, which by implication can be paid less attention).

S11 expert: The most important thing, if they want the common [look and feel], is being able to apply styles.

S6 expert: You use [the GET request] just for testing to see if the system is up and running. The important thing is the handle POST request.

Finally, the expert helps to shape the newcomer’s model of the code by **sizing code**. Very often, this occurs implicitly when the expert ends a chunk of explanation. This tells the newcomer that there is nothing else of significance contributing to the functionality under discussion, and thus they can assign a measure of its size. This is often a cause of surprise for the newcomer, indicating that the code’s structure is unexpected and that knowledge transfer is taking place. In particular, code that is smaller and simpler than expected often generates a reaction of surprise.

S6 newcomer: That’s the end?!
S6 expert: That’s it.

S8 expert: (scrolling through code)
S8 newcomer: Oh, that’s reasonably short.
S8 expert: Well, it’s not too bad.

S1 expert: Let’s look at undoable command. (opens UndoableCommand interface, which is very short)
S1 newcomer: (laughs)

S3 expert: I think we’re done.
S3 newcomer: It’s huge!

Taken together, these activities in the sessions provide a rich map of the current code, using abstraction to group, name and layer the code into a comprehensible structure, and developing a sense of its characteristics. However, this ‘map’ of the code is only one aspect of the mental model; three other aspects are still to be discussed.
8.1.2 How it works - an Algorithmic view of code

To explain how code works, experts match the code to the feature it implements and provide a view of the typical path of execution through the code. Newcomers are able to infer some of this information from the code itself, so this aspect of the discussion tends to be highly interactive, with the newcomer checking their assumptions with the expert.

Mapping code to its responsibilities

Experts provide a mapping between areas of code and the functionality they implement - i.e. the code’s responsibilities within the system.

S3 expert: (demonstrating software) If you click Pay All, that’ll bring you to the payment entry screen where you can type in how much money you want to pay off the case.  
... 40 minutes pass  
S3 newcomer: (opens MakePayment.aspx)  
S3 expert: That’s the page you saw for entering the credit card information.

For code that does not directly implement an observable feature, experts can still give its responsibility.

S5 expert: There’s a nice error message later on when we get a packet that has some data points that are valid but the key doesn’t [...] match anything that we know about and that would be the responsibility of this thing.

S8 expert: [This server] is going to be in charge of authentication so that we can technically kick you out if you stop paying your bills.

S7 newcomer, in interview: That was really important to understand; how this tool is extracting this information? [...] What are the parts of the code that are responsible for this implementation?
Implementation details

Having identified the responsibility of some code, the expert may go into more detail about the inner workings of the implementation. By describing the typical flow of execution through the code, they give the procedure or algorithm that has been developed to meet that responsibility.

The implementation is broken into steps, and the amount of detail for each step varies: sometimes a core line or variable or object will be described in careful detail, but other code comprising many lines can be grouped and summarised in one phrase (another example of abstraction). However, newcomers report that it is not generally useful for the expert to go through every line of code at this level; only for unusually complex code is this worthwhile.

S6 expert: (Points at getOrCreateNode method) This is where we create a node, so first of all it takes in a (points at ID parameter) an ID. So first of all we look in the cache. (Points at idToNode variable) ID to node is the cache to see if we’ve already got the node, if we’ve already got it we don’t need to create it [...] so (points at ”if (node == null)”) if it doesn’t already exist we have to go and ask the database..

S3 expert: Basically what we’re doing is, for each case in (points to caseIDs variable) case IDs, case insertion. So there’s a call into (selects TempTakeOffHoldCaseIDs table name). What we do is we’ve got these temp take off hold case IDs, we have a temp table, and we’re sorting all the case IDs into that temp table. We end up with a big long fat string (selects INSERT string) that will effectively insert all the IDs that we’re interested in into this temp table....

S4 expert: (Opens graph_fetch function) Graph fetch, it basically just says, load a graph from the database, gimme all the (points to ‘context’ dictionary) bits I need out of it, and then (points to ”render_to_response” method call) render the JavaScript that represents that graph.

In no session did participants use the debugger to step through the code. This is surprising, given that some newcomers reported that they would make heavy use of the debugger for individual exploration. This is discussed more in Section 8.4.4.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

Newcomer interaction

The newcomers’ development skills allow them to interpret some of the code on first viewing. They frequently check their understanding by asking questions or making statements about the code, especially when the expert is presenting an Algorithmic view.

S1 expert: Let’s look at undoable command. (Opens UndoableCommand interface) This is just the regular command interface.
S1 newcomer: Gotcha. So we can execute, it has a label, and it undoes. Sweet, ok.

S4 expert: This action also listens to the dashboard for changes to a chart’s timescale.
S4 newcomer: Oh right, things being sent up from the backend when someone loads a chart.
S4 expert: Exactly.

Experts also work to prevent misconceptions on the part of the newcomer, by highlighting plausible but incorrect models of the system.

S7 expert: I’m just connecting different parts and it doesn’t add another edge, it just changes the existing one.

S5 expert: We build up this map of the keys (selects self.keys=newkeys line) and then we swap it out. We don’t edit the map that’s already there.

S4 expert: The view and the edit page... already have the (selects script element) JavaScript for the graph built into the page. They don’t have to go back to the server to edit it.

Experts do not describe the workings of all the code, instead focusing on either the ‘core’ execution flow through the system or parts of the code that they were recently involved with. Where the newcomer had a task, some experts were able to focus on code relevant to that task. This issue is discussed further in Section 8.4.2.
8.1.3 Why it is there - a Rationale view of code

While a skilled newcomer could eventually reconstruct the architecture of unfamiliar code, its purpose may be more obscure (depending on the domain) and the reasons for design decisions may be unrecoverable. During the sessions, experts pass on the requirements of the system and the reasons for the design and implementation choices.

Passing on requirements

In many sessions, the newcomer was somewhat familiar with the general requirements of the system. When the requirements were unfamiliar, the most common way to communicate the purpose of the system was to demonstrate the system in action, from an end-user’s viewpoint.

S8 newcomer, in interview: If [S8 expert] wants to set up a call and then run through their UI showing what happens, that’s where you begin because then you understand what the actual goal is.

The newcomer needs to know the business rules that the software is intended to implement; where this is not obvious from the domain or the demonstration, experts often retrieve or sketch diagrams to explain what is to be done and why the processes are required by the end users.

S2 expert: (drawing diagram of stock transit) So you despatch something, so you know there was one thing on that lorry. That lorry might take four days to get there, so... you won’t necessarily be able to match that to the things that arrived.

S7 supervisor: [The system] allows the architect to specify their as-designed architecture, then map it onto the actual as-implemented architecture of the system... and then they get feedback on where things are inconsistent, so (points at diagram on slide) that’s the process there.

S3 newcomer, in interview: How to read that [business rules] flow diagram... that’s essential, if you can’t read that you can’t navigate the system. That session was more of a very very high level, this is what the system does as a whole.
Occasionally, experts also characterise the requirements in terms of the clients’ wishes to provide more context for the requirements. This allows the newcomer to predict the clients’ response to changes in the deliverable.

S3 expert: (indicating logo) That’s just, [the clients] like to see their logo on it.

S2 expert: [The support team] like having XML files because it’s quite easy to check, especially if you’re on a remote connection or something.

S2 expert: (describing changes previously requested by client) At some point [the client] might go, tell you what, we just want [to treat] all of Europe as one country. So I tried to make this as reusable as possible.

Overall, requirements and context are passed on in sessions by word-of-mouth. Other than the occasional diagram, not a single session made use of formal requirements documentation, and no newcomer requested this form of documentation.

Reasons for design choices

The business requirements provide context for the features of the system, but this does not dictate the internal design of the system. There are many possible ways that the system could be implemented, and the newcomer can work more effectively if the reasons for the implementation choices are understood. If experts are not forthcoming with reasons, newcomers will use the session to extract reasons for unclear choices.

S3 newcomer, in interview: A lot of it would be trying to get the thought process behind why something was being done. There’s no point just knowing exactly what something does and that be that, it’s better to get into the mind of the person that actually did it in the first place cause that’ll help you understand a lot of parts of the system. So a lot of it would be why did you do that, why is it done this way.
S12 newcomer, *in interview*: There’s two issues in understanding the code: One is understanding the proper technology, and the other is understanding why was it implemented this way and not another way. Maybe there is a reason why this was done that way [...] If it’s a more critical decision, for example a business requirement or a maintenance problem or something specific, then it’s very hard to know why.

Design choices are in evidence throughout the system, from the highest level architecture down to the use of variables. The highest level structure of the code is often presented in terms of an architectural pattern as described in Section 8.1.1; it is generally assumed that the newcomer is familiar with the generic reasons for choosing a pattern such as Model-View-Controller. Other design principles, less clearly named, result from the requirements and domain.

S3 expert: It’s all one hundred percent data driven, basically the system can’t do anything unless it records the fact that it’s doing it, so it’s very data centric.

S12 expert: Whatever is in here (indicates project] depends only on the Eclipse core classes, it doesn’t refer to language, so it could be used for C or Java... this is language independent because you can use Eclipse to develop in Java, C and a couple of other languages.

Many of the design choices become *project conventions* and newcomers will be expected to adhere to these in their work. Pointing out such conventions often goes hand in hand with explaining the reasons for them.

S11 expert: This string functionality is for the localisation. We call a function and pass in a const value, so we’ve no actual real strings anywhere in the code.

S3 expert: All the URLs to the actioners are in here, so if I ever have to move the folders around or whatever, I change it here. S3 newcomer: Ok, that’s good, everything’s in one place. It’s easy to manage. S3 expert: Yeah, that’s the idea.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

S5 expert: This keysync thing is a full-blown thread, because it needs to talk to the database and we really don’t want any of that stuff blocking while it’s working.

Non-obvious design choices often have their roots in problem workarounds or unexpected business rules. Without knowing these reasons, the newcomer may be tempted to ‘correct’ code in ways that lead to regressions. Pointing these out in the session avoids this unproductive activity.

S6 newcomer, in interview: [S6 expert] had a piece of code in there which generated these time IDs manually. So I thought he’d made a mistake, I thought he just didn’t realise it was in the library and [I used] the library function. I ended up getting errors related to time UUIDs later that I didn’t understand, and when I asked him it turns out he’d come across the same error and realised the function that was provided by the library wasn’t compatible with that particular database, that was then the reason he’d implemented it manually.

S8 expert: (pointing to ‘TMP’ variable) Temp is nonsense. Basically, this curl (points at curl call) will give out nothing, but we have to define a variable so that Asterisk will have something to log it out to.

S2 expert: [This customer] will have a GB record and then an IE record. It’s the same customer record in our systems, but the reason is that Irish stuff is going to be in Euros.

Other non-obvious choices are artifacts from changes in the design. The reasons for the choice no longer stand, but it would be too difficult (or not worthwhile) to change the code. Understanding this, the newcomer will not waste time searching for currently valid reasons for the design, nor use the previous design as a template for future work. This temporal issue is discussed further in Section 8.1.4.

S3 expert: [One of the projects] started in use as web applications in there but now we put them into C development. S3 newcomer: (opens project from Recent Projects list) Why isn’t it, or why hasn’t it been moved?
S3 expert: We wouldn’t move it because it would have a lot of relevant references to development, common shared stuff.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

This quote also illustrates the newcomer’s interest in design choices that do not immediately make sense. Newcomers try to figure out the reasons in the sessions, but will resort to asking ‘why’ when necessary.

S6 newcomer: Why are you supporting [both POST and GET]?
S6 expert: The only reason I’m supporting GET is so I have an easy way of finding out if something is still running.

S3 expert: [This] is our business object, and [...] we don’t want to store it in the session [...] 
S3 newcomer: Why wouldn’t you store it in the session, just as a matter of interest?
S3 expert: The reason we don’t store it in the session is because a user might have two windows open in the same session and we manipulate two different sets of payments.
S3 newcomer: With two different session IDs?
S3 expert: They have the same session ID.
S3 newcomer: Oh, same session ID, ok.

The expert will sometimes provide a reason in the form of a short anecdote. These stories typically relate to problems and how they were overcome. In addition to providing background for design choices, storytelling also transmits advice about best practices and avoiding pitfalls.

S2 expert: He was trying to do a lot of this stuff in the database without having any support from the client app, but now ... we’re taking it on.

S3 expert: You can see the header here’s got test all over it, because there were issues in the early days where people were on live [system] where they thought they were on test and vice versa.
S3 newcomer: That’s a good idea, we had that problem with [previous project].

The discussion of the reasoning behind design choices and guidelines is not out of historical interest, but often forms part of the expert’s advice for working on the code. This is described further in Section 8.3.
8.1.4 What is changing - a Temporal view of code

The majority of software is under active development or maintenance; as a result, the codebase that the newcomer must learn is not a static artifact. The newcomer sees a snapshot of the code which might already be out-of-date and is likely to contain confusing artifacts of previous and planned development. Fortunately, the expert is able to provide guidance on past, present and future changes over various timescales.

Ongoing changes

As active developers, experts are typically aware of longer-term ongoing changes to the codebase. Informing newcomers of these activities helps them to make sense of apparently inconsistent code.

One category of ongoing change is the temporary fix. Such fixes are expected to change in the near future, and being temporary workarounds, they are also poor examples for the newcomer to follow. The expert may justify the poor practice with a reason for the necessity of the fix.

S2 expert: I’m just hard-coding a run at the moment because they didn’t have it at the time.

The codebase may also contain dead code; that is, code that is no longer executed but has not yet been removed. Pointing out dead code is valuable if the expert gives a reason for why the dead code is still in the codebase; this provides some insight into the team’s practices, and allows the newcomer to avoid wasting time on comprehending code that is no longer in use. Encountering dead code may also lead the expert to plan future changes to remove it (again passing on knowledge of how the code is likely to change in the future).

S6 expert: (points at JSON code) That code used to get called when we were able to navigate through the graph.
S6 newcomer: Oh, yeah.
S6 expert: So it’s no longer... but we’ll put it back in under certain circumstances, like if someone’s holding shift down we might want to do a reload.
Most importantly, the project’s project conventions may have changed over time, resulting in two or more conventions coexisting in the codebase. Project conventions include coding standards, code naming conventions and guiding principles that guide the implementation choices. The newcomer needs to know which of these conventions is the current one to be followed. Experts point out these differences and often give the reason for the change in convention.

S11 expert: In the last project there was a lot of work done to change over the server code and to pull the business logic out of different places to make beans, and some areas of the server are more and better beanified than others.

S2 expert: This .NET app is prior to EasyLog so the log is in a specific location relative to the project, whereas the EasyLog method that we use later actually talks to a service.

**Planning changes**

Experts are also aware of planned changes that will alter the code, and this impacts the session; they point out areas that are about to change, including areas that are marked for removal and therefore not worth explaining in detail. The timescale of the changes can vary from “expected immediately” to “in the next version” or at an unspecified time.

S5 expert: Don’t mind this (points at Aggregator class) aggregator thing at the top. This was intended to be our one central source for, given a set of data, all the different aggregation that we can do on it, but... all of that will go away later.

The changes may be already designed, in which case the expert is able to describe exactly how the code is likely to change (including the impact on its complexity).

S6 expert: When we add timestamping, that’s gonna create a new level of complication, but I don’t think it’ll be anything else that’ll be different, so there’ll be a new method to retrieve between timestamp A and timestamp B.
Alternatively, the expert and newcomer may collaborate on designing the changes. This often forms part of the expert’s advice for working on the code (Section 8.3).

S4 newcomer: Is this the point where we’d want to be parsing the spec and converting it to your object oriented representation?
S4 expert: Yeah. So this could be something like (types “var json=json”)
S4 newcomer: Ok, fine, and then we start to process that.
S4 expert: (Deletes typed line) For targeting the JSON thing we do this, and then for... yeah, all the same. It should be relatively contained to just (indicates for-loop) this bit for logging and saving and whatever.

Discussing these changes provides a set of provisional updates to the model: areas of code that will appear, disappear, work differently, or change characteristics.

Temporal issues were most common in S1, S4, S5, S6, S8 and S11, all of which were under active development with the experts working on code as recently as the day before. They were rare in S2 and S3 because the code was not under active maintenance at the time of the explanation; the codebases were relatively static during the sessions and planning future changes were the main source of temporal content.

Software developers are often quoted as saying that “the code is the one reliable document about the system” but this shows that although the code may be the best source, it is not 100% reliable due to its constantly shifting nature. Additional information is still required: what other developers are working on or have recently changed, and whether the version in source control is truly the most up-to-date code available.

This would seem to represent a difficult task: in trying to understand the visible code, the newcomer must also remember which parts are out of date, unused, or likely to change, and why, and who is working on them. However, this is information that all team members must maintain awareness of in order to work effectively, so trying to explain only the most current version of the software and documentation might not be desirable even if it were possible.
8.1.5 How experts represent code to newcomers: Summary

Taken together, the four views on the code (Structural, Algorithmic, Rationale and Temporal) provide a comprehensive representation of the codebase that matches the realities of software development. The resulting mental model of the code incorporates not only the current snapshot of the code, but the reasons for its present implementation and the current and future changes that can be expected. Much of this information cannot be gleaned from the code or documentation, so the sessions represent an important channel of communication from expert to newcomer.

This section has focused on the expert’s representation of the codebase, but software development is about more than just code. In the following sections, the scope widens to incorporate other topics communicated during the sessions.

8.2 How experts support newcomers in locating information: Shaping newcomers’ information seeking strategy

Experts do not necessarily have all the answers to development (and environment setup) problems, but instead have well-developed strategies for finding the answers. Thus, they are rarely blocked by missing information. Newcomers arrive with general strategies for software development information seeking, but during the sessions, experts supplement this with project and team specific strategies so that newcomers will be able to make independent progress on development work.

While the first research question focused on the code, this question has a broader scope, and considers code, code artifacts, documentation, reference materials and co-workers as potential information sources on which the expert provides guidance. This section examines the session activity to answer the second research question: how do experts support newcomers in locating information?
8.2.1 Establishing terminology

Project-specific terminology allows the developers to communicate more clearly. Experts provide explanations for domain and project terms:

S12 expert: In [this project] you have Java model (opens JavaModel.java) which is the source model. It’s the implementation of the implementation model from the core package. I’m just going to refer to [that project] as the core
and [this project] as the Java implementation.

S8 expert: [This server] is what handles dub dub dub and authentication, but [that server] is the application server that’s...
S8 newcomer: Ok, ‘dub dub dub’?
S8 expert: WWW.
S8 newcomer: I see, ok.
S8 expert: You’ve never heard it called ‘dub dub dub’?
S8 newcomer: People call it different things. I call it wobbly. (Laughs)
S8 expert: (Laughs) Wobbly? Ok, that’s a good one.

Initially, this vocabulary is a barrier to the newcomer’s information seeking. Until newcomers adopt this vocabulary, they have difficulty phrasing their questions to experts and interpreting the answers. This is discussed in Section 8.4.4.

8.2.2 Documentation

Project documentation is important to newcomers, especially if they are entirely new to the code. Although minimal documentation is considered acceptable in small projects, a project lacking any documentation is a problem for newcomers. Newcomers who have had some prior involvement with the code are less concerned by incomplete documentation.

S3 newcomer, *in interview*: [The project] was very well documented as a project for a starting point. There was... the flow diagram that took up nearly a wall and a half of space. It’s a really good reference point even now, you couldn’t navigate through the system without it.

S7 newcomer, *in interview*: I don’t think the code is very complex. It’s just that there’s no documentation at all, no comments at all. So that would be complicated, just to give the code to someone.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

S6 expert: We have the wiki to go with the project and the bug tracker to some extent, and the source code [laughs]. It’s fairly minimal but for two people it doesn’t make sense for us to write any more. It’d become outdated and anyway there’s no one there to read it yet.

In most sessions, experts introduce newcomers to the documentation by showing them wikis or diagrams. Occasionally the newcomer is provided with documentation to read before the session.

A major problem with documentation is that it may be out of date. When code or development practices are changed, documentation updates are typically the last part of the process (if they are made at all) so areas of ongoing change are frequently undocumented or the documentation is out of date. Understanding these discrepancies is important for preventing confusion; however, this is difficult to discover from the documentation itself, limiting the value of individual study of the documentation. During the session, experts provide a general sense of which areas of documentation are up-to-date.

S6 newcomer: (Scrolls wiki sideways to see long installation commands)
S6 expert: I need to update this documentation. You won’t need yfiles or any of that stuff.
S6 newcomer: Ok.

S11 expert: The wiki is good... but some of the earlier architecture stuff is a bit out of date, because some of the connection stuff’s changed and hasn’t been followed through.

Rarely, planned work may be described in the documentation but not be implemented in the code. In one case, the newcomer had read about a feature in the wiki prior to the session, but discovered from the experts that it had not actually been implemented. This is an unusual example of documentation being ahead of implementation instead of behind, and it was equally confusing to the newcomer.

Like the code, the documentation is also affected by temporal issues such as versioning. Unclear versioning also impacts the utility of unguided documentation use.
S6 expert: These instructions [in the wiki] are still necessary if you want to run the old version, and they’ll work for the old version.
S6 newcomer: Yeah, it’s worth keeping them there for the moment.

In a large, long-term project, it is possible for the documentation to become so large that the newcomer needs guidance just to locate the relevant areas. In S10, experts warn newcomers about this issue and offer assistance.

S10 newcomer, *in interview*: For whatever section you’re working in, you probably need someone to tell you, look, this is the part of the wiki to look at, but you just put an email out to the team and they tell you where in the wiki.

While documentation is valuable, there is a widespread sense that knowledgeable co-workers are a better source of information.

S10 developer: I was going to mention this wiki, this is one of the most useful things, apart from [S10 expert]! (laughs)

Put together, the experts are providing the newcomer with a strategy for using the documentation: where it is, whether it is up-to-date, and how to seek help with navigation. With this strategy, the documentation becomes a valuable asset; without it, newcomers are liable to take documentation at face value, hindering their understanding of the project.

### 8.2.3 Co-workers

For the newcomer, co-workers (including the expert and other newcomers) can be valuable sources of information. On smaller, newer projects (such as S1 and S6), the expert and the newcomer(s) comprise the whole team, so no additional co-workers are mentioned. In contrast, projects taking place in large organisations typically mention many co-workers in various roles, and long-term projects are more likely to reference previous maintainers. In the sessions, the experts match co-workers to their areas of expertise, place limits on their own knowledge, and indicate the availability of other developers.
Co-workers are sometimes introduced in person, when the expert calls on them to answer a question as part of the session. Occasionally a co-worker will overhear the session and offer input. More often, the expert will mention other developers by name or role as good sources for a topic. This serves several purposes: as well as indicating the co-worker’s area of expertise, it also (implicitly or explicitly) indicates the limits of the expert’s knowledge.

S2 expert: (Addressing co-worker) You used to support this, didn’t you? How often would you get a customer that didn’t have a name, and how often would that be a problem?
S2 database expert: It shouldn’t be a problem because the data comes from the customer database.

S8 expert: I’d have to look at the [project] code (navigates directories)
S8 developer: I can chime in here because this is my code.

S11 expert: I haven’t worked on SPA at all so I don’t what’s there. [Co-worker] would be the man for that.

The latter quote also illustrates that experts give the limits of their own knowledge. This helps newcomers to avoid wasting their time with queries that the expert cannot answer (either because the expert does not know an area, or has forgotten it). Some further examples of this:

S9 expert: The identifier is basically source, target, compilation unit and offset. Or actually, it might not need (points at SOURCE on diagram) this. I don’t remember exactly, you can just see it in the code.

Experts are able to characterise the kind of help that the newcomer can expect from co-workers.

S11 expert: [Co-worker] has implemented a refresh button here... so he’s going to have a conference call to tell us how it works, and he’ll say, oh, it’s in the wiki and here’s the link.
In the sessions, newcomers are often encouraged to come to experts for help.

S11 expert: You know where I am. You can come over anytime and we can go through that.

When experts are unavailable, other newcomers from the session may form part of the newcomer’s information seeking strategy.

S12 newcomer, in interview: In all the demos of the tool, they had already set the model to show architectural violations as warnings, and when I ran the tool I wasn’t seeing any architectural violations even though I forced them because it was defaulting to ignore. Because that hadn’t been mentioned, I had to ask [another S12 newcomer] about that and she told me.

A major difficulty with co-workers as information sources is that they may not be available to answer questions at all times. From experts, newcomers learn about co-worker availability. This may be implicit especially for developers no longer on the project.

S10 developer: This [wiki] is one of the most useful things, apart from [S10 expert]! (laughs)

The awareness of co-workers’ knowledge, limits and availability provides the newcomer with more strategies for information seeking.

8.2.4 Reference materials

Aside from teaching novices (as in S1), reference materials are useful to developers for third-party code such as libraries, APIs and frameworks. By pointing out the use of third-party libraries in the code, experts implicitly let newcomers know that knowledge about that code is more likely to be found in third party documentation.

S9 expert: (points at IJAVAELEMENT on diagram) IJavaElement is the Eclipse interface for that.
S8 expert: This is all done with Tornado’s built in asynchronous HTTP client class, which means that you can create arbitrary HTTP requests throughout your request handlers.

This effect can also be seen across multi-team projects, where components provided by other teams may be considered ‘third-party’.

S11 expert: As application developers we use an awful lot of the components that are provided to us by Infrastructure [team], like all the Sage components.

As with the documentation, experts help newcomers locate reference materials and give their opinions on the quality of the content.

S8 expert: [Third party tool] is nice... there’s a good wiki for all the Russian documentation and dodgy English translations.

S12 expert: It is described as a single linked list. You need to read the Java [documentation] because it also allows you to navigate both ways, it’s just the other one is very ineffective.

Although third party reference material may be an important part of day-to-day development, discussion of these sources represents only a small part of the information seeking strategy. This is probably because newcomers are already equipped with general strategies for locating this kind of material, and will benefit more from a guide to internal, project and team specific sources.

### 8.2.5 Code and code artifacts

The newcomer needs a strategy for locating code within the project. The mental model of the code, discussed in Section 8.1, forms an important part of this strategy. In particular, the Structural overview described in Section 8.1.1 provides the newcomer with one strategy (following the abstract, high level view of the architecture) for locating code. Other strategies, such as the use of IDE search and navigation tools, are part of a developer’s standard repertoire and are rarely discussed in sessions, but
these become more effective as the newcomer learns the project’s terminology and project conventions.

Code artifacts (such as log files and sample input and output data) may also be sources of useful information to the newcomer, but their locations are not always obvious. Experts occasionally show newcomers where to find these artifacts and provide strategies for interpreting them.

S10 developer: It’s easy to tell, if your [internal tool]’s wrong, if you just search your log file for “initFID”.  
S10 newcomer: “InitFID”.  
S10 developer: If you see “initFID”, there’ll be a line telling you you’re wrong.

S2 expert: (Types path.) There it is. (Opens directory.) Flash file. (Opens example output file.) So there’s a one line header (points to top line)...  
S3 expert: All this is information that’s in the XML file, so I’ll take a brief aside and show you an XML file. (Dictates path) [...] When you open it up, it’s a zip file.  
S3 newcomer: Oh, you can have multiple PDFs.  
S3 expert: Yes, and your XML is what defines everything else, so this is all the information we parse.

Newcomers value the location of code artifacts because they can provide vital resources and clues for development, but are difficult to locate, with even experts occasionally struggling to locate them.

S10 newcomer, in interview: You need to know what logs to even look at [...] You’re gonna have your software logs, you’re gonna have your database logs, you’re gonna have your logs then for your actual [project] so you need to look at the trace files because if things are going wrong you’re gonna have errors in there. But you need to know what type of error or exception or whatever, and you can blindly go looking but you need to know, if it’s a database thing there are certain things you can look for.

S2 expert: You won’t find the [output file] location of any of the files in the config, which is a shame [...]

131
S2 newcomer: Might be worth adding one as an artifact in the solution itself, for testing?
S2 expert: That’s a good point.

Providing the newcomer with a strategy for locating code is a major purpose of the overall session, but experts typically spend little time on the location and format of code artifacts.

### 8.2.6 Overall strategy

In addition to advice on the various sources of information, the expert also helps the newcomer judge which source will be most effective for answering a query.

S10 newcomer: Does anyone follow [wiki discussion pages]? No?

S11 expert: There’s a little tool in the browser [...] It’s like a Parsley debugger, and you can see the messages being fired around, and you can see who fires and who catches them and things like that
S11 newcomer: Oh cool. [...] Sounds really great.
S11 expert: It is, but the quantity of messages that are being fired around can be a bit overwhelming in terms of trying to find the one you’re interested in.

### 8.2.7 How experts support newcomers in locating information: Summary

In the sessions, the experts pointed out a number of different information sources, including documentation, reference materials, co-workers and code artifacts. Each of these sources had advantages and disadvantages, and experts sometime gave their opinions on the quality of a source. In doing so, experts helped to shape the newcomers’ generic information seeking strategy into a more effective project-specific strategy that would otherwise have required a longer process of trial-and-error to create. The newcomers struggled to make use of information sources without the correct terminology, so establishing domain and project terms also formed part of this shaping effort.
8.3 How experts support newcomers in making contributions: Giving advice about working

In addition to representing the code and supporting the newcomer’s information seeking, experts also support the newcomers in making contributions. The third research question concerns the nature of this support.

Figure 8.3: Giving advice about working
8.3.1 General advice for efficient working

Over time, experts develop a set of working practices that are specific to a project. In particular, they can recognise and follow its conventions, handle common problems with established workarounds, and predict when long-running tasks will complete. Without these working practices, newcomers are liable to waste time on problems that the team has already solved. In the sessions, the expert gives the newcomer advice for working on the code, either in person or via documentation, and this builds the newcomer’s system skill.

The expert’s experience allows them to predict long-running tasks. This allows them to detect malfunctions and to manage multiple tasks efficiently. Experts inform newcomers about typically long-running tasks.

S6 expert: You can run that [build] in the background, that’ll take a while to do.

S3 expert: If there are tons [of items in the list] it can take thirty seconds plus to load.

S11 newcomer, in interview: Starting the server [...] could take five minutes or it could take three minutes or it could take a minute, and if you start the Flex client before that’s started, you won’t get a connection because it’s not up. [...] It’s timing, it’s all those little gotchas.

Experts pass on their workarounds for common problems and error conditions (including gotchas). This prevents the newcomer becoming blocked by a problem that has a known solution.

S8 expert: If you wrote output to the incoming request, you would have to call self.finish, or else the request will hang and 504. That’s one non obvious error.

S6 expert: Sometimes the [software] just doesn’t reply, and you don’t know where the problem is, so at least if you run this [diagnostic tool] you know that the servlet’s up.
Newcomers are familiar with the general principles of software development tools such as IDEs, but might not be familiar with the particular tool or version used by the team, in which case experts pass on advice for tool use. Additionally, there may be project-specific interactions between a familiar tool and the quirks of the code. Expert advice helps the newcomer use the tools with skill.

S2 expert: With going to view definitions on this [method] especially in this version of .NET, if it’s overloaded you might not go to the definition that you think you’re going to [...] so if you’ve got loads of overloads, check your parameters to make sure you’re looking at the right one.

Experts point out project conventions to ensure the newcomer is aware of them. In addition to aiding the location of code, these guidelines and philosophies continue to influence the maintenance of the code. The newcomer needs to absorb this knowledge in order to make code changes in line with the original design.

S3 expert: It’s a singular wrapper for complex SQL interactions. Just so there’s no SQL anywhere in the project other than here.

S11 expert: The biggest rules are, we want no ActionScript in here, if at all possible... and we want no text in here.

Sometimes, the expert’s advice is accompanied by a story; a short anecdote that illustrates the reason for the practice and the consequences of failing to observe it. As described in Section 8.1.3, stories usually describe issues encountered by the team before the newcomer joined, helping the newcomer to avoid repeating previous mistakes.

S10 developer: That’s actually a really important email that a lot of people ignore [...] An email will come from [coworker], right, and then invariably at least one of the twenty people will send me an email afterwards saying ‘my sandbox is broken, my build won’t work’, I log onto their machine and go ‘did you update [the internal tool]’ and they go ‘no’, so always there’s at least one person.
While some projects share **ownership** of all the code, others divide ownership of certain areas. This may be a formal division between teams, or an informal arrangement according to expertise, preference or authorship. Where these divisions exist, the expert will inform the newcomer, either implicitly or explicitly, which areas of code are off-limits. Implicit advice may take the form of describing the difficulty of working on an area, without providing strategies to overcome it. In terms of ability to work on code, this might appear to have a negative impact on the newcomer’s self-sufficiency, but knowing when to defer code changes to another developer is an important skill in such teams. As such, this information is most useful to the newcomer when accompanied by a **reason** for the ownership practices.

S5 expert: For example, you can’t write the PID file until you have daemonised, but you don’t want to daemonise until you’re certain that you can write the PID file.

S5 newcomer: (laughs) Gotcha.

S5 expert: [...] Also you have privileges, and logging, yeah, there’s a lot of annoying gotchas in all.

S11 newcomer, **in interview**: I’ve worked out what [new code] needs to go in, but as to [...] what I’m allowed to change, I’m not too sure so I have all those questions, because you have an [infrastructure development team] who are meant to look after that stuff. It’s not meant to be the application developer that’s meant to be changing that code.

The general advice from the expert helps the newcomer assimilate the team’s accumulated knowledge of workarounds, conventions and typical behaviour. With these skills, newcomers can work more effectively on the code.

### 8.3.2 Task specific advice

In S2, S5, S10 and S11 the newcomer had been assigned a first task before the session, and in S1 the expert went through the assigned task as part of the session. In these cases, the expert can usually provide task-specific advice, such as **recommending changes** and **predicting issues** with the change.
S1 expert: When we undo we animate the move to the source location. Now, this might actually be a problem with this stuff that we’re planning to do because if we want to move more than one node then we might just not animate it to keep it fast, otherwise it might be really slow. So that’s something you could consider.

S4 newcomer: Is this the point where we’d want to be parsing the spec and converting it to your object oriented representation?
S4 expert: Yeah, so this could be something like (types example code) [...] It should be relatively contained to just (indicates for-loop) this bit for logging and saving and whatever.

As part of the task-specific advice, the expert may also point out code that can be ignored because (perhaps despite appearances) it is not relevant to the newcomer’s first task.

S1 expert: We don’t need to think about that [event driven system], it’s just so the UI gets updated.

S11 expert: That’s all you need to worry about from a viewpoint of navigating here.

In teams with strong code ownership, the newcomer’s task may not be in the expert’s area. In this case, the expert can recommend another developer for advice.

S11 expert: [Co-worker] sits next to me, you can talk to him about [your task].

S11 newcomer, in interview: [The first two sessions] pointed me to code, where stuff was, which was a good starting point because at least I had a sample. [...] The third walkthrough was the actual person who was in my area of code, which was a bit more, targeted. That was done over the phone. He just pointed me to the actual classes I’d be working on, what needed to be changed, what didn’t need to be changed.

Without access to test data, newcomers may be unable to run the software, reproduce an issue or test their changes. Experts are able to
provide the newcomer with the same test data that they use. This is also an example of the expert shaping the newcomer’s information seeking strategy with regard to code artifacts (Section 8.2.5).

S3 expert: There’s a special test [credit] card number that I have to root out. It’s in the documentation somewhere. The test card number allows you then to just put in any amount of money and it’ll say yeah, that’s been successful.

S11 newcomer, in interview: I was looking around and I [realised] ‘hey, I don’t have any data’ and [the database expert said] ‘oh, you’re pointing to the wrong database, change that now’. I wouldn’t have known, I would have been there forever.

8.3.3 How experts support newcomers in making contributions: Summary

In the sessions, the support offered by the experts was partially dependent on whether the newcomer had been assigned a development task. If so, experts were able to offer specific advice for that task. In either case, experts offered general support and helped the newcomer to assimilate the team’s shared knowledge about working on the codebase. This project-specific advice helps newcomers to apply their general development skills to the specific project without becoming blocked by problems that have known workarounds.

8.4 Problems with onboarding: Session difficulties

Onboarding sessions do not always run as effectively as they could. Both experts and newcomers can experience difficulties that impact the session. This section addresses the fourth research question by describing reported and observed problems that reduce the value of onboarding sessions.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

8.4.1 Session configuration

The configuration of the session can impact its effectiveness. This section examines the impact of three roles that are present in computer-based onboarding sessions.

Roles

The configuration of the session can be described by examining who performed each of three roles, and when and why the roles shift during the session.

Driver: has control of the keyboard and mouse (i.e. operates the
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

computer).

**Director:** decides which files to open and what code to examine (i.e. directs the content of the session).

**Owner:** the main user of the machine outside of the session.

The most common configuration, seen in seven sessions, was that the expert took all three roles, leaving the newcomer with a relatively passive role (though some newcomers remained highly engaged). However, this was not the only configuration observed.

- In S1, the expert directed and drove, and then handed over to the newcomers for some development, all on the expert’s machine. From this point the newcomers proceeded as a pair with occasional prompting from the expert.

- In S3, the expert directed throughout, and the newcomer drove for most of the session, using the expert’s machine.

- In S6, the newcomer and expert each drove their own laptops. Initially the expert directed, helping the newcomer to set up the development environment and giving a high-level overview of the code. Gradually, the newcomer began to explore independently and the expert’s role changed to answering questions from the newcomer.

- In S9, the session revolved around a set of diagrams drawn on a whiteboard.

- In S12, the expert directed and had one of the newcomers drive (using the newcomer’s laptop) so that the code under discussion was visible on the projected screen for the benefit of the other participants. Later, the newcomer briefly directed and drove in order to demonstrate some code changes. As the session moved towards planning the next code change, the expert resumed directing.

The driver role

Sessions S3, S6 and S12 were unusual in that the newcomer was driving. This was not deliberately negotiated, but it just “made sense” to conduct
the session that way. Most other newcomers reported that they had never experienced onboarding sessions in which the newcomer was driving, and suggested that this was more likely to be encountered in training sessions or in very short, ad-hoc explanations at the newcomer’s desk.

When asked about session configurations, participants identified a variety of pros and cons and personal preferences. Having the newcomer drive was regarded as better for newcomer engagement because the newcomer cannot ‘switch off’ during the session. This in turn improves their ability to find the code on their own, and is a type of learning-by-doing. It prevents the expert going through the system too quickly for the newcomer to follow, and provides the expert with continuous feedback on how well the newcomer is picking up the system. Finally, this arrangement was considered better when the newcomer had specific questions to ask.

S6 newcomer, in interview: It’s probably better for the person who doesn’t know what they’re doing to be the person controlling, like [S6 expert] could probably show me the project and fly through the packages and I mightn’t... being forced to do it, I’d imagine, forces you to pay more attention even at one level, just that you look at what you’re doing.

S12 newcomer, in interview: It will be [the newcomer’s] responsibility to know how to look around, and it would just give an impression maybe of how that person thinks, it would demonstrate, did they understand what the expert’s thinking, as opposed to them driving the experience, the novice just saying yes to everything, they walk away and there’s nothing happening.

However, having the newcomer drive was regarded as frustrating for both expert and newcomer, because the expert has to direct the newcomer through ‘back seat driving’. If the newcomer is having difficulties navigating, this may indicate that they have not yet acquired the high-level overview. The session would be slower and less fluid, and experts may have difficulty locating code due to their inability to take small actions to establish location within a file. The latter is considered more likely to be an issue if the expert does not have a high-level overview of an area of code.
S4 newcomer, in interview: The expert knows precisely where [a piece of code] is in relation to all of the other lines and patterns and shapes of the code around it [...] If the expert is not exactly sure where they are, then they can’t just scroll up by a couple of lines to check something and go “oh yes it must be below here”, and then skip down by three pages and immediately find it. They can’t take any of these little investigative actions that will help them drive. I think it just leads to lots of confusion.

S5 newcomer, in interview: If [the newcomer] hasn’t looked at [the code] at all, maybe it’s more useful for the experienced person to control it, because you want to send them to a specific area to look at instead of scrolling up and down through stuff and [there is a] control issue with it being annoying when someone is “no no up, no down, no, stop, gimme the computer”. (Laughs)

This navigation issue was seen in S3 and S6, but the newcomers in those sessions still preferred that configuration:

S3 expert: So if you go back up to the reference, to that guy.
S3 newcomer: To which one, er?
S3 expert: To where it loads.
S3 newcomer: To where we came from? (switches to another tab)
S3 expert: Oh no, it’s back in that file.
S3 newcomer: (switches back to first tab)
S3 expert: Control and scroll to go back to where you were.
S3 newcomer: (returns to large case statement)
S3 expert: There you go.

S6 expert: Wanna scroll up, it might be there?
S6 newcomer: (scrolls down)
S6 expert: I think that might be it actually, there’s a, maybe two variants of that, if you (points to top of screen) scroll up again?
S6 newcomer: (scrolls up)
S6 expert: (points to code) So we handle post, ok, so initialise, yep, so scroll down a bit.
S6 newcomer: (scrolls down)
S6 expert: Yeah.

Opinions varied on how to choose a session configuration. Some participants were adamant that the expert should always drive; others
that the newcomer should always drive. Others suggested more nuanced possibilities; a common suggestion was that the expert should drive initially and provide a high level overview, and then hand over to the newcomer whose ability to locate code would have been improved by this information.

S4 newcomer, in interview: The expert is the one who is driving... I feel like that ought to be a general recommendation, I think letting the newbie drive is not useful.

S12 newcomer, in interview: I think the expert should probably drive initially, definitely on the high level, and then the novice should drive it from there.

Finally, personal preference came into play.

S3 newcomer, in interview: It would work in different ways for different people. We’ve since had some people work for us that... they would rather you showed them, and they would sit there and watch whereas other people prefer hands on and prefer to be guided towards something and find it themselves, I would be the latter... I wouldn’t say you could make a rule, the person who owns the system should never use the mouse, I think it would be too strict that way, I think it would depend on the person.

S11 newcomer, in interview: I think [having the newcomer drive] is ideal... [but] personally I think them showing me initially and recording and stuff is probably better, and then go look at the code myself.

The director role

In all but two sessions, the expert took the director role throughout, controlling the focus of the session. The exceptions were S1 and S6. In both of these sessions, the newcomer took the director role towards the end of the session. In S1 this was explicitly negotiated and corresponded with a deliberate shift in the session from explanation to newcomer development. In S6, this occurred gradually; the newcomer began to look at code outside the expert’s direction and ultimately took over the
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

director role, with the expert providing commentary and answers as the newcomer explored.

S6 newcomer: (Scrolls down to goDeeper method) I’m at this. (Indicates comment about recursion into nodes) [reads comment] (Invokes tooltip on nodesToEgoIDs parameter) Ok, so you recurse...
S6 expert: So that method, is this goDeeper again?
S6 newcomer: Yeah. (Scrolls up to top of goDeeper method definition)
S6 expert: Oh yeah, ok so (Points at goDeeper method implementation)
S6 newcomer: You draw itself iterating until? (Selects depth parameter and invokes tooltip)
S6 expert: If we’re going three levels deep, this can’t recurse more than three levels. (Points at depth<=maxDepth test) That’s what this is testing.

The newcomer was able to take this role because of two unusual attributes: knowledge of the intended design of the system, and the shared owner and driver roles due to the two-laptop session configuration.

Many of the comments in Section 8.4.1 are based on an assumption that even if the newcomer is driving, the expert is still directing (leading to inevitable backseat driving). While this is generally the case, S6 demonstrates that this is not necessarily a requirement of an onboarding session.

The owner role

The third role in the sessions is that of the owner of the machine. Usually, the owner of the machine also drove it, an exception being S1 where the newcomers pair programmed after some instruction from the expert. This quote hints at the strong link between the owner and driver:

S4 newcomer, in interview: I have to go off and ask the expert, and we come back to my desk and I sit down, because it’s my desk, and they lean over and go oh yeah, up a bit, down a bit, over a bit...
Using the newcomer’s machine in the session, as in S6, has its pros and cons. The main advantage is that the newcomer receives expert help in setting up and using the environment; this helps to iron out setup problems that would hinder the newcomer’s independent exploration. This is frustrating for the expert, because environment difficulties prevent the session from proceeding smoothly (as described in Section 8.4.3), but a smooth session may leave the newcomer with unaddressed problems afterwards.

S6 expert: Test fails?
S6 newcomer: Yep. (scrolls text in Console tab)
S6 expert: Ok... so you’re failing because of the connection to [server], so don’t worry about that.
S6 newcomer: Yep, that won’t work, ok, fine.

Additionally, the expert and newcomer may prefer different toolsets, making it difficult for them to use one another’s machine. This is less of an issue in companies that standardise the environment for all developers.

S5 newcomer, in interview: [S5 expert] uses a completely different IDE to me, he uses Vim, and he’s happy to go through that and I’ll use my thing [...] and neither of us really can get past the barrier of like, God, what does this thing do [...] That could be a barrier in places that don’t have a standardised toolset.

Session configuration as a tradeoff

Overall, session configuration can be seen as a tradeoff between optimising the session for the expert (who is under time pressure from other work and is more valuable to the team in the short term) and optimising for the newcomer (some expert frustration is acceptable if it allows the newcomer better engagement).

S11 newcomer, in interview: [Having the newcomer drive] is ideal, but I suppose when people are under time constraints doing deadlines and they’re trying to give you a handover too, it’s kind of it’s a tradeoff.
S4 newcomer, *in interview*: There is a tradeoff there between how annoyed you want the expert to be and how much of their time you want to use, versus how much time you want the newbie to spend stumbling around, but I feel like the newbie can do this stumbling around later, direct targeted questions to the expert and you’ll waste less of their time.

For the expert, the optimal session gives the newcomer information as fast as possible, which they are expected to consolidate in their own time while the expert returns to more important work, and so the expert takes the driver, director and owner roles. For the newcomer, the optimal session gets them past early roadblocks and allows them to make contributions early on in the onboarding process, and so the newcomer is the owner, perhaps the driver, and has the potential to take the director role.

### 8.4.2 Session timing

The timing of the session within the wider onboarding process has an effect on its utility. Sessions given before the newcomer has been assigned a task are less likely to be relevant, and delays afterwards reduce the newcomer’s ability to retain information from the session.

#### Task and relevance

In five cases, the newcomer(s) had been assigned a task prior to or during a session, and were are able to use the session(s) to learn more about that task (highlighted in Section 8.3.2). However, where the newcomer had no task at the time of the session (e.g. S6), or where the task was outside of the expert’s area of expertise (as seen in S10), newcomers often reported that the session was not relevant to their eventual first task.

S6 newcomer, *in interview*: He gave me an overview of the entire two projects, but it ended up that the part that interested me was a very small subset of that. The work that I’m doing is a separate project that intersects in a small part.

The lack of a task is an indication that the session may not cover content that would most help the newcomer to contribute.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

Time until modification of code

Typically, the newcomer attempted to work on the code within hours or days of the session. However, the delay between the explanation and the modification varied, from S1 where the newcomers began to add a feature during the session itself, to S3 where the newcomer was delayed by six months and S12 where one newcomer did not expect to work on the code for at least a year. Long delays are a difficulty for the newcomer because without any contact with the code, they are liable to forget much of the session’s content.

S3 newcomer, in interview: I think one thing we realised was that doing [the session] at such an early stage, before there was going to actually be any serious work done on it, was not the best idea. It was a good six months before I was even reintroduced to the system [...] Basically we had to do [the walkthrough] again when it was time for me to kick back into it.

S12 newcomer, in interview: I don’t remember anything [from the session]. I won’t be working on the software for a year, by which time it will all be different.

This issue is related to the presence or absence of a first task for the newcomer. In the observed sessions, newcomers who had not been assigned a task were more likely to be diverted onto other work after the session, delaying them from consolidating their knowledge through code modifications and making the session a less effective use of time for all involved.

8.4.3 Newcomer’s development environment

In a software development context, the development environment includes shared resources (such as a source repository and build machine) and each developer’s workstation(s). For the developer to work effectively, the workstation must be set up with software such as IDEs, source control and build tools, and each of these must be configured correctly. This represents a challenge to the newcomer: until their development environment is set up, they are unable to explore code independently or contribute changes, and initially they struggle to set up the environment themselves.
Importance of the development environment

A development environment is obviously required before the newcomer can contribute to the project. Less obvious, however, is the value of independent exploration of the code immediately after a session in order to consolidate their knowledge. In particular, newcomers describe the use of debug tools or log statements to trace the execution flow.

S11 newcomer, in interview: What happens a lot of the time is you’re told to set up your environment and people’ll go walk through the code on their setup and you don’t even have a running setup so you can’t even test what they’ve just told you.

S5 newcomer, in interview: I just don’t pick up on learning if someone just shows me a block of code. Ok, it does some stuff, I want to see what it actually does. […] I have put in log statements with dummy words doing specific stuff [resulting in] a page of stuff and then I can trace the order of it.

Without a working development environment to enable this, the session content begins to fade from memory. As a result, a working development environment is a high priority for newcomers.

S11 newcomer, in interview: Setting up the environment, that’s always one of the hardest things to do at the very start. […] That’s crucial, so I almost think someone should be given a hand setting up their environment, just to go through the little gotchas, make sure it’s ok.

S6 newcomer, in interview: It’s a good idea to set it up on your own machine. Even just checking the project out and making sure it works. It might work perfectly well on [the expert’s] machine because that’s what he’s using to develop but I could have some quirk or some thing that doesn’t make it work and that would slow you down later on.

Difficulties of environment setup

Newcomers struggle to set up environments without help due to project-specific settings and resources.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

S11 newcomer, in interview: [The expert is] invariably going to come in with everything running on their machine already pre-existing, so they’re not telling you about databases, they’re not telling you about config files, and they’re not telling you about gotchas and stuff you can fall over on.

S6 newcomer, in interview: I think it’s a good idea to set it up on your own machine, even just... checking the project out and making sure it works. It might work perfectly well on his machine because that’s what he’s using to develop but I could have some quirk or something that doesn’t make it work and that would slow you down later on when you’re trying to get it to go.

Additionally, the newcomer may be unable to set up an environment without first acquiring rights and permissions. These prerequisites include workstation administrator rights (to permit software installation), software licences, a source control system account and so on. The newcomer in S11 had particular difficulty with environment setup despite the corporate use of sandbox environments, due to missing rights, accounts and licences:

S11 newcomer: I don’t have a UNIX login, but I have [co-worker]’s. The thing is, I have no admin rights so I couldn’t install [the new internal tool]. I’ve just got admin rights so I can install that, so that’s what I was doing just before I came in. Once I have that I think I should be able to go through the code and debug it.

In contrast, the newcomer in S8 was provided with more privileges than required:

S8 newcomer, in interview: They gave me access to the code [...] I don’t know if they assumed that I would need to modify it or not, but they gave me the modification rights anyway.

Setting up an environment is an infrequent task, and experts occasionally struggle to help because they have forgotten the details.

S6 expert: (Opens Preferences dialog) Preferences.
S6 newcomer: (Opens preferences dialog)
S6 expert: Uh... (browses amongst options) Oh, where is this? (cancels Preferences dialog)
S2 expert: It might even be the fact that [this project] needs to be on a different platform. I might have thought it was a .NET 1, tried to build it in .NET 1, wouldn’t build.

As with much documentation, the setup instructions are often outdated, and newcomers sometimes update them based on their experiences.

S6 newcomer, in interview: The first task was just to check out the code and get it compiling and running on your local machine. I made a few changes to the wiki that have changed since the time [S6 expert] wrote it, so now if a stranger come along hopefully they could at least get that part.

Alternatively, there may be no setup documentation at all, leaving newcomers to make assumptions about the correct environment.

S12 newcomer, in interview: If there was a specification of the exact dependencies, this is what the tool is being built and tested against, that would have been useful. I initially started developing with the tool against Java 7 because that’s what was installed on my laptop. I have since reverted to Java 6 because that’s the minimum language that’s required to run the tool.

Finally, the newcomer requires an up-to-date copy of the source code in order to contribute, but the combination of an actively maintained system and an unfamiliar source control system make this difficult. Out-of-date code caused confusion in S6.

S6 newcomer: If you toggle the overlay, does that change a variable on the server side?
S6 expert: It did, and that wasn’t working very well so I changed it. It now takes the value as well.
S6 newcomer: So it sets it... (points to toggleOverlayVisibility call, which does not take a matching value)
S6 expert: Oh, so sorry, you have yesterday’s code which does the toggle, which isn’t appropriate. Now it takes the value as well.
S6 newcomer: And sets the variable rather than setting?
S6 expert: Yeah, so it’s not toggling.
S11 newcomer, in interview: How to pull down [code] and make sure I had the most recent updates, they were the kind of key points.

Going from one to two developers might be a particularly difficult case: the original developer may not be using full version control, hindering the newcomer’s efforts to locate up-to-date code.

S6 newcomer, in interview: He’d sent me a snippet of code that he’d used but hadn’t checked in.

Factors in environment provision

While a development environment is a high priority for the newcomer, its priority for the expert depends on the session configuration (discussed further in Section 8.4.1). If the session involves the newcomer’s workstation, the newcomer’s development environment becomes a higher priority to the expert because the session cannot proceed without the ability to view the code (although other important development tools may not be used in a session). This was the case in S6 in which the first half of the session was devoted to setting up the newcomer’s environment. However, the setup was not completed and the newcomer was not able to build or run the code (only to view it) by the end of the session.

S6 expert: So, [first project] and [second project].
S6 newcomer: (Opens the two project download pages in different tabs)
S6 expert: They need to be in sister directories. (Scrolls wiki)
S6 newcomer: Ok, first one. (Clones first project using git)
(Clones second project using git)
S6 expert: When that’s done, you’ll run Maven package inside the [first project] directory.

(Some time later, the newcomer and expert discover that a network problem is preventing a full build on the newcomer’s machine.)

S6 expert: Ok, so don’t worry about that.
S6 newcomer: That won’t work, ok, fine.
S6 expert: We’ll sort that out again. Ok, so we’ll start, we’ll go through the codebase then.

151
In contrast, most other sessions were conducted on the expert’s machine with an existing environment, and little if any session time was devoted to the newcomer’s environment. In these cases, the newcomers were largely expected to drive this process themselves, using their knowledge as developers and project documentation where available.

S11 expert: You have [source control] and all that set up, do you? You have a [development environment], do you?

S12 newcomer, in interview: I pretty much got the code running myself before I started. The code was available on SourceForge, I just downloaded it and I installed the dependencies.

This can be seen as another factor in the tradeoff between optimising for the expert (in which no session time is wasted on ‘administration’ tasks) and optimising for the newcomer (in which the expert ensures the newcomer is not blocked by missing resources). However, this may be a short-term optimisation only, since a newcomer with a working environment is able to make more efficient use of the sessions.

As with many onboarding activities, the newcomer can apply their general development ability (in this case, setting up a development environment), but will be more effective if this is supplemented with project-specific details: guidance on exactly what to install, where to find it and how to configure it for development on this project. Depending on the project’s complexity, this can be a difficult task for both newcomers and experts, but a working environment is important for effective learning outside of sessions and for contributing to the project.

8.4.4 Content delivery

The delivery of the session’s content has a major impact on its effectiveness. This section examines the impact of six aspects: the use of diagrams, the expert’s knowledge of the code, the project terminology, the session plan, the use of debugging tools, and low-level examination of code.
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

Diagrams

Diagrams were used in six out of the twelve sessions for communicating a variety of concepts: business requirements, system architecture, computer science concepts and the inner working of concepts. The majority of diagrams were sketched by the expert during the session, but three were previously existing diagrams used as a project reference.

S9 expert: The cross reference has two (extends XREF box in sketched diagram) important things in here. It’s just a collection, it’s basically a source (adds ‘SOURCE’ to box) target (adds ‘TARGET’ to box) and (draws line under TARGET) that’s the most important thing.

S2 expert: We had a document about this, hang on. (Browses, opens Visio diagram showing overview of data flow)

As described in Section 8.1.1, newcomers value diagrams for the abstraction they provide. When a session lacked diagrams, newcomers usually searched for diagrams afterwards, asked the expert to provide a diagram, or mentioned in the interview that diagrams would have improved the session. The most requested diagrams are a high level overview of the whole system and a more detailed representation of a smaller, task-relevant area.

S11 newcomer, in interview: When you go searching up in the wiki you’ll find the [diagrams] alright. They have the dumbed down, million miles in the sky version where it just shows you the technologies that have been used, and then, once you get to the different sections, you start finding the other, more targeted versions.

S4 newcomer, in interview: I’ve lost half a day to not understanding something and eventually talked [S4 expert] into making me a diagram. [...] I had to ask him in the diagram, please put in the precise file and function names, because otherwise I will not be able to follow it, and he did, and then it helped. I can go and look up that precise thing and see how it was linked together, but without that very precise information I was not able to do this.
Alternatively, one newcomer generated diagrams from the source code using the Doxygen tool:

S12 newcomer, in interview: I had a look at the code briefly, and I generated a UML diagram to see roughly how the code laid out. [...] It can give you a good idea of the overall structure and you can see what components are calling what other components or what methods are calling other methods. It’s useful but it has its limitations as well.

However, diagrams are not automatically helpful to newcomers. Where the diagram’s conventions or context are not familiar, the newcomer struggles to make sense of the information.

S9 newcomer: I just understood badly because of the UML thing. I thought those were attributes and that you were...
S9 expert: They are attributes.
S9 newcomer: Those are attributes?
S9 expert: Yes.

S12 newcomer, in interview: [The expert’s diagrams were not useful because] he had implicit knowledge that I wasn’t party on. [...] If I was there I’d have to ask some questions on what they’re talking about, because I wasn’t really sure of the context or the significance, because I haven’t written any of the code myself. So no, I didn’t find that very useful.

The latter quotation is interesting because other newcomers in that session, who had more understanding of the context, reported that the diagrams were useful.

Finally, newcomers value diagrams as a post-session reference, and leaving the session without any was a source of regret:

S8 newcomer, in interview: [S8 expert could have improved the explanation by] insisting that I leave with a diagram, or take some notes. But that’s entirely my fault and as the more experienced person I should have known this.

Diagrams are a valuable reference for both general and specific software comprehension, and from these reports it would seem that they
are underused in onboarding sessions. However, the context and the conventions must be understood before the newcomer can make use of the information they provide.

**Expert’s knowledge of code**

It is common for the expert to have an incomplete knowledge of the code. This can be for a number of reasons: the expert wrote the code and has now forgotten how it works, or the expert was not the original author and has never known how parts of it work.

S7 expert: It’s model view controller, basically, so it’s the back link from, er, controller to the model, or is it, er, I don’t remember.

S2 expert: Is that a DTS package or a job? I can’t remember, I think it’s a... guess it’s a job.

A knowledgeable expert’s explanation typically makes heavy use of abstraction. (Section 8.1.1) Abstraction requires a thorough understanding of the area of code under discussion; without this, the expert cannot present an abstract view of it, and instead resorts to low-level just-in-time comprehension which a skilled newcomer would be able to do independently.

S5 expert: It doesn’t start that [thread], I think it just loads it but doesn’t start it. (Scrolls up in code) Oh, maybe it does start it immediately. Oh, I see, no, it creates an instance of this but it (points at start call) doesn’t start it until the [service] itself has been told to start.

While experts can recover some information from the code (as can the newcomer), a lack of knowledge about the code’s origins impacts their ability to deliver the Rationale view, because this requires historic knowledge of design decisions and tradeoffs.

Lacking knowledge about the code, the expert may be unable to provide a definite answer to the newcomer’s questions. Instead, they may offer a guess, a partial answer or refer to a knowledgeable co-worker. Guessing is
acceptable when it is made clear that it is an uncertain answer, but it is preferable to use the opportunity to influence the newcomer’s information seeking strategy.

S11 newcomer: Is there a way to prioritise jobs? Jump them up in the queue?
S11 expert: I’m not sure. Some things have to be run in certain orders and I know there’s an object, there’s a task group and there’s an atomic task group [...] Some things can be moved and some things can’t.

Some experts recognise this problem and state that they would refresh their memory of the code before giving a walkthrough. In this aspect, temporal issues also affect experts and so they may pay particular attention to recently changed code. Comments in the code are valuable to experts during explanations and updating the comments was a common preparation technique amongst those experts who prepared.

S6 expert, in questionnaire: I documented the code and put comments to describe complex code in order to make it easier for myself to remember.

However, many experts did not prepare for the session, and the resulting difficulties made it harder for newcomers to establish an accurate mental model of the code.

Terminology

As described in Section 8.2.1, developers on a project use project-specific vocabulary to communicate. However, newcomers are often unable to adopt these terms quickly enough to support the questions they wish to ask in the session. As a result, questions are held up as the expert tries to interpret the newcomer’s question and the newcomer struggles for the correct terms. In extreme cases, this discussion takes so long that the original question goes unanswered. This is a problem for both the newcomer and expert, but ultimately the newcomer suffers until the communication issues are resolved.

S11 newcomer: That chart, is that, what, uh... I’m getting confused now between the terminology, there’s Parsley...
S11 expert: Is it Sage, is it?
S11 newcomer: Sage, that’s what I’m thinking of.

S7 newcomer: A question: you clicked on architectural views, before that?
S7 expert: Architectural views?

[...]
S7 newcomer: Ok it’s not architectural view, then, it’s...
S7 expert: Well, the only architectural view is this. (Selects Architectural Relations tab)
S7 newcomer: Yes, that’s why I’m confused.
S7 expert: It’s a view in terms of Eclipse views, this is an editor and that’s a view. It’s not an architectural view in terms of a logical view. It’s just a list, right?

S12 newcomer: That source, that Java model is shared among the implementation?
S12 expert: Java implementation.
S12 newcomer: Java impl- sorry (laughs) there’s many concepts to get.

The main source of confusing terminology was the internal terms used for codebase concepts, such as ‘architectural view’ in the second quote. Other sources, such as domain terms, language constructs or technology stacks, caused occasional problems, but these differ from the internal terminology in that there are generally recognised definitions for those terms outside of the project. For internal terms, definitions can only be found within the project, if at all, and newcomers’ reported experience of the domain did not help them to avoid this issue.

Adopting the project’s terminology is key for effective project communication and information seeking, but this difficulty impacts the effectiveness of the session.

**Session planning**

In some sessions, experts state a plan at the start, indicating that they have given the walkthrough some thought. In S10 the expert brought a previously written list of topics to ensure the session was comprehensive. Another common organisation strategy was to follow an important execution flow through the code.
S1 expert: Yesterday [I implemented] the automatic layouts on the graph, so we can bring those up and take a look at how those work. Then I’ll go through the source code for the fully automatic stuff, and then we’ll look at how we can make those handle [undo and redo]. That’s basically the plan.

S8 expert, *in questionnaire*: No preparation. I walked them through a typical request flow, using message passing diagrams to show how various components interacted.

However, other sessions are less organised, with experts explaining code ad-hoc.

S7 expert, *in questionnaire*: No preparations, just go with the flow.

Tangents are common in sessions, triggered either by the newcomer’s questions or code that reminds the expert of areas to discuss. In some sessions, tangents are discussed immediately, which complicates the overall session plan but ensures that content is not missed. In others they are deferred so that the explanation does not deviate from the plan, but this carries the risk that the newcomer’s question will never be answered.

S4 expert: Now you mention it, I’ll just skip over to something else we’re doing. (searches for code)

S2 expert: [That is] the merge step (points at diagram) So, if I finish taking you through the actual client app, then I can actually show you how that merge happens.

Sometimes, the expert reached the end of the walkthrough and then cast around for more information to give to the newcomer. Experts can feel under pressure to deliver as much information as possible in a session.

S3 expert: That’s really it. The only other thing I could show you briefly is-
S3 newcomer: That’s about the fourth time you’ve said that. (Laughs)
S8 expert: Ok, I’m trying to think if there’s anything else that needs to be shown off.

S5 expert, *in questionnaire*: Some mild concern that I’d bumble through it, not explain anything of value and end it after only a couple of minutes. Experience has shown me that I rarely do this, but it’s still a recurring fear.

However, a more common problem for the newcomer is the sheer amount of information delivered in a session. Newcomers suggest that this can be mitigated by taking breaks in the explanation, and through self-study immediately afterwards (development environment permitting).

S3 newcomer, *in interview*: If someone is just explaining something to you for one or two hours solid, it can all get a bit jumbled up. So I did kind of expect that [S3 expert] would try and break it up and put it into manageable chunks of information.

S11 newcomer, *in interview*: The first thing I did [after the session] was actually went in and had a look at [the code] myself. I went through to see if I could remember what [S11 experts] had said to me, [to] rejog my memory.

**Use of the debugger**

Debugging facilities (such as the ability to set breakpoints, step through code and examine variable values) are a common feature of current IDEs. Many participants identified the debugger (or related debugging techniques) as a useful individual exploration tool:

S6 newcomer, *in questionnaire*: [To understand unfamiliar code I would] run it, add logging statements, run it again.

S8 newcomer, *in questionnaire*: [To understand unfamiliar code I used] insertion of various debugging statements throughout a piece of code...
S10 developer, in questionnaire: [To understand unfamiliar code] I used a debugger to check in more detail what was going on. The debugger was very useful for getting a clear idea of what was going on.

This makes it surprising that debugging tools were not put to use in any of the sessions. Some sessions did not have debugging tools available; for example S5 which used the Vim editor, or S9 which centred on a flipchart rather than a computer. However, in the majority of sessions, the participants used an IDE such as Visual Studio or Eclipse to navigate around the code. Thus, debugging tools such as breakpoints were available, but even when the expert was following the flow of execution through the code, these tools were not used. The newcomer in S10 highlighted the potential for the use of debugging tools within explanations:

S10 newcomer, in interview: [To improve on the explanation, the experts could have] run a live debug session. I think it’s the fundamental approach. [...] A little diagram and a debug session going through that, and then [for each] hit in the debug session just point [out where] each one is in the diagram. It’s easier to relate to.

Low-level examination of code

In some sessions, experts went through code line-by-line. This level of detail is not of great value to newcomers during sessions; as experienced developers they are able to understand code at this level of detail through independent examination when needed. Low-level explanation of irrelevant code contributes to information overload. Unusually complex code is an exception where expert explanation can be valuable.

S3 newcomer, in questionnaire: In general I’ve found that a full on, line by line approach proves to overload the newcomer. An approach of high level explanation followed by questions derived from working with the code has been successful in the past.

S2 expert, in questionnaire: Nothing is useless, even going through the code line by line can be useful for certain
code-bases or functions. But the most useful actions are the higher level or broad descriptions first, and then to get more and more detailed.

S12 newcomer, in questionnaire: The expert explaining every line is probably overkill for an experienced newcomer. Going through complex code is useful, but not all code.

8.4.5 Hard-to-find information

Newcomers report that within the context of their projects, certain pieces of information are difficult to find without expert guidance. Some of these are described in previous sections:

- The meaning of project terminology (Section 8.4.4), without which the newcomer struggles to interpret domain and project specific terms. This can result in difficulties having technical conversations with experts.

- The location of test data (Section 8.3.2) necessary for exercising code and testing modifications.

- How to set up a development environment (Section 8.4.3), including how to gain necessary permissions and how to set configuration details.

- The reasons for design decisions (Section 8.1.3), especially for unusual implementations. Newcomers may spent time puzzling over this code, or repeat previous implementation mistakes, without knowing the reasoning behind the design.

In addition to these problems, newcomers may struggle to find information related to gotchas, unclear names and delocalised code.

Gotchas

Experts have become so at home in the system that they forget to mention ‘gotchas’ - quirks of the codebase or the environment with known but non-obvious workarounds. Gotchas may occur as part of environment setup, development, or running the code. The expert’s advice
(Section 8.3) typically includes workarounds for some gotchas, but others can hold up the newcomer unnecessarily.

S11 newcomer, *in interview*: It’s very hard for a person who’s been working on [the system] for the last year or two years or six months to expect them to remember the gotchas, because they’re not gotchas in their heads any more. They’re stuff they don’t even think about any more because they know it and they don’t even realise they know it, because it’s so ingrained in their head.

Unclear names

Experts usually discuss the project conventions and some important code entities, but undocumented and unclear names are a common cause of problems for newcomers. The lack of supporting information makes it difficult for newcomers to rule out competing hypotheses about the purpose of the entity.

S12 newcomer, *in interview*: The [variable] name didn’t reflect what it could do. He explained [that in] different contexts, it’s hard to name it, which I understood then. But why not add some comments? Why not give some kind of indication [of the context]. Is that a database connection? I don’t know. Is it some kind of edge?

S8 newcomer, *in interview*: The [server] names give you absolutely no hint as to what it might do. (Gives examples of server names, which are taken from a well-known work of fiction.) Each of these ones could be the critical front end user facing timing sensitive application and I wouldn’t have any idea.

S12 newcomer, *in interview*: [The problem is] just the name [of the variable]. I didn’t make that link logically that that was what it meant [...] As I was looking at the code I kept coming back: is that what it’s meant to do? And because there’s no comments either saying that this is what it’s representing... (Laughs.)
CHAPTER 8. GROUNDED THEORY OF ONBOARDING SESSIONS

Unclear or inconsistent names also make it more difficult for the newcomer to match their model of the architecture to the code.

S4 newcomer, in interview: I went to [S4 expert] with a very general question of I don’t understand where the M, V and C bits of this are [...] I’m modifying the model and the view in the same function, and I know that’s not how he designed it, but it worked. Because I don’t know where the other bits were or even what ***** files they were in because they’re all named the same and differently and, oh, goodness.

Delocalised code

The comprehension process is made more difficult for the newcomer by the distributed, event-driven nature of many codebases. In these cases, newcomers often request diagrams showing the interactions between parts of the system (see Section 8.4.4).

S4 newcomer, in interview: There were plenty of different files that were tied together in strange ways, and there were events and listeners and it really wasn’t clear... it still isn’t perfectly clear how it all fits together. [...] It’s very frustrating, I’m not sure that a better explanation could have been given for that code without a diagram. It’s awkward, particularly with the [functionality] splitting over lots of different files. Many of them are very short, they’re just tiny snippets; that’s six lines, this is twenty lines, and the purpose of each one and the order of execution is very unclear.

S3 newcomer, in interview: [Newcomers to this system] would find it hard to grasp the interconnectivity of everything. There’s a lot of different processes going on, simultaneously, there’s stuff that’s scheduled to happen out of hours, there’s five servers that are running all the time and they’re all equally important.

8.4.6 Problems with onboarding sessions: Summary

This section identified five aspects of onboarding sessions that affect their value: the session configuration, the session timing relative to the
newcomer’s first task, the provision of the newcomer’s development environment, the content delivery techniques employed and the handling of additional hard-to-find information.

8.5 Summary

The theory presented here provides insight into the first four research questions, describing the typical onboarding session, the variations that were observed, and the difficulties seen and reported by both experts and newcomers.

In Figure 8.5, the components of the previous four sections are combined, revealing the overall structure of the theory. By providing three types of session content (influenced by the task where there is one), experts can boost newcomers’ development of four important resources that are important for effective newcomer onboarding. The process can be affected positively and negatively by four session attributes, highlighting potential difficulties for both experts and newcomers.

Having identified these difficulties, it is possible to make recommendations to address them in future onboarding sessions, and this is the focus of the following chapter.
Figure 8.5: Newcomer-expert onboarding session (NEOS) theory
Chapter 9

Recommendations for Practitioners

The theory identified a set of problems encountered by both newcomers and experts during onboarding sessions (Section 8.4), and also indicated some practices that work well.

To answer the final research question ("How can onboarding sessions be improved?"), data from the sessions, background questionnaires, interviews and evaluation questionnaires were examined for recommendations that might address each of the problems, and for actions that were consistently considered worthwhile and should be maintained. These recommendations are described in this chapter, and the final section discusses ways in which they could be evaluated.

9.1 Session timing

The timing of the session, relative to the newcomer’s first task and their readiness to work, influences how useful it is to the newcomer.

- Sessions should take place after the newcomer’s task has been assigned. Without a task to provide focus to the session, experts may feel under pressure to explain everything, which tends to overload the newcomer with information (Section 8.4.4). Instead, discussing the newcomer’s task within the session provides some structure for the expert and ensures that the session is immediately useful to the newcomer (Appendix G.2.4). Having no task may also indicate that
the newcomer is not expected to work on the code in the immediate future; in this case a session is of low value and should be postponed until a task has been assigned (Section 8.4.2).

- **No code should be explained in a session until the newcomer’s development environment has been set up.** This can be achieved either by holding the session after setup is complete, or by beginning the session with environment setup. Newcomers value the ability to consolidate the knowledge from a session immediately afterwards - this helps to mitigate problems of information overload (Section 8.4.3), while delays exacerbate it. The recommendation to use the newcomer’s machine for the session also helps to bring environment problems to light.

9.2 Newcomer’s development environment

Newcomers report that development environment setup is often a challenge.

- **The expert should help the newcomer with development environment setup.** Environment setup can be a considerable issue for the newcomer (Section 8.4.3) and delays impact not only their productivity but their ability to make the most of the session. Particularly problematic are project-specific details including permissions, configuration, location of test data, settings for running the system and so on. These impact the newcomer’s ability to install development tools, examine source code, build the software and exercise specific code of interest (Section 8.4.3). This help can be delivered through up-to-date documentation, comprehensive automation, and in person.

9.3 Session configuration

Session configuration, described in terms of the driver, director and owner roles, affects not only the content of the session but the ability of the newcomer to proceed effectively afterwards. There is a tendency for the expert to take all three roles, which, while more convenient for the expert,
does not tend to uncover and address the newcomer’s issues. The following session configuration recommendations apply where the expert and newcomer are able to use each other’s development environments - where this is not the case, there is less scope for freely assigning roles (Section 8.4.1).

9.3.1 Choice of driver

- **At the very beginning, the expert should drive;** this is because the newcomer does not have an overview of the layout of the system, so navigation is difficult and the expert would need to back-seat-drive if the newcomer drove (Section 8.4.1).

- **The expert should use this time to provide a high-level overview of the system** so that the newcomer is able to locate files by concept (Section 8.1.1).

- **When the newcomer has a high-level overview, they should take over the driving role from the expert.** This achieves a number of things: the newcomer gains practice at navigating the system, and is able to slow down the session to their own pace, preventing the expert from going too quickly through the system (Appendix G.2.4). This recommendation is also supported by the original participants in their responses to the ‘fit’ questionnaire (Table E.31).

9.3.2 Choice of director

- **In general, the expert should take the director role during sessions,** unless the newcomer has significant project knowledge (Section 8.4.1).

9.3.3 Choice of owner

- **The session should always be conducted on the newcomer’s machine -** i.e. the newcomer takes the owner role. This recommendation is made for two reasons. The first is related to the
recommendation that the newcomer should drive as soon as they are able. There is a strong link between the owner and driver role, so having the newcomer take the owner role is expected to result in the newcomer also adopting the driver role by the end of the session (Appendix G.2.4).

The second reason is to address the issue that, without a working development environment, newcomers are unable to consolidate their knowledge after the session or work on contributions to the project (Section 8.4.3). By using the newcomer’s machine for the session, environment issues are more likely to be encountered and fixed. As a result, although the session might feel ‘interrupted’ by these tasks, the newcomer is able to make use of the session content immediately afterwards (Section 8.4.1). Newcomers value this assistance highly (Appendix G.2.4). This recommendation is also supported by the original participants (Table E.32).

9.4 Session content and delivery

The session content and its delivery is a major factor in its value. Experts are often unprepared to lead sessions and do not make the best use of the time available, while information overload is a major problem for newcomers.

- **Experts should review code prior to the session.** Experts often believe that this is unnecessary because they know their code well (Appendix G.2.4). However, forgetting the details of code in sessions is common (Section 8.4.4). Techniques used by experts to prepare include reviewing the overall structure of code, reviewing recently changed areas, and adding comments (Section 8.4.4).

- **Experts should provide diagrams of the high-level structure of the system early in the session.** The high-level structure is valuable to newcomers because it allows them to navigate the system and locate code, and newcomers will seek it out if it is not provided (Section 8.1.1). Diagrams are valued for their abstraction, avoiding information overload, and newcomers commonly request structural
information in diagram form (Section 8.4.4). The high-level overview is among the most useful information a session can provide, because navigating the system and locating code are challenges for newcomers (Appendix G.2.4).

- **Experts should provide detailed diagrams of task-relevant structures.** As described earlier, discussing the newcomer’s task in the session is highly recommended (Appendix G.2.3). Newcomers value task-relevant diagrams; in contrast to the high-level structure diagrams, task-specific details such as names of specific files or functions make the diagram a more useful reference. However, such diagrams can be misunderstood; the newcomer also needs to understand the diagram’s context and conventions (Section 8.4.4).

- **The session should include discussion of the newcomer’s development task.** When sessions take place without reference to the newcomer’s task, newcomers often report that much of the content was not relevant to their eventual task (Section 8.4.2).

- **Experts should trust newcomers’ development skills.** As experienced developers, newcomers are able to reconstruct how code works, so going through code line-by-line is not necessary and is not a good use of session time (unless the code is unusually complex) (Appendix G.2.4). Too much detail contributes to information overload (Section 8.4.4).

- **Experts should help newcomers to locate answers to questions that the expert cannot answer.** Guessing answers is acceptable only if it is clear that the expert is guessing. It is often more useful to the newcomer to provide guidance on where the information could be found (Section 8.4.4). In particular, introducing newcomers to knowledgeable co-workers is a valuable activity (Appendix G.2.2).

- **Experts should keep sessions short.** Information overload is a major problem for newcomers (Appendix G.2.4) but experts can feel under pressure to use the full length of the session to explain code (Section 8.4.4). Sessions should end as soon as the newcomer has enough information to attempt their current development task. If
this is not possible, long sessions should include breaks, during which the newcomer has the opportunity to review the information (Section 8.4.3).

- **Newcomers should make notes and keep reference materials.** Sessions typically provide the newcomer with useful information including the locations of files, names of code elements, project-specific terminology and diagrams (both pre-drawn and sketched during the session). These are valuable post-session references and newcomers regret not taking notes or keeping copies of these materials (Section 8.4.4), adding to the information overload problem.

### 9.5 Hard-to-find information

Some of the information passed on during sessions is particularly valuable to newcomers because it would be difficult to find otherwise (especially in projects that lack strong documentation cultures). However, experts often neglect to pass on valuable information. This includes the environment setup details described earlier.

- **Experts should define terminology before using it.** Without project-specific terminology, newcomers struggle to frame their questions about the system and understand the answers (Section 8.4.4). Particularly troublesome are terms with multiple meanings and/or technical meanings outside the project that do not match their meanings in this scope. Newcomers value these definitions in the session (Appendix G.2.4).

- **Experts should provide solutions for ‘gotchas’.** Gotchas are problematic quirks of the codebase or environment with known but non-obvious workarounds (Section 8.3). Unless experts pass on the workarounds, they can slow newcomers down unnecessarily as they spend time investigating gotchas and rediscovering known solutions. However, experts often forget to pass on gotcha solutions, either because they are one-time operations that are easily forgotten, or because they have become second-nature to use (Section 8.4.5).
Documenting gotcha solutions at the time of discovery may be of value to subsequent newcomers.

- **Experts should provide the meaning of unclear names.** Unclear names (for anything from variables to servers) hinder newcomers in their attempts to understand the code independently, especially where comments are sparse (Section 8.4.5). While project naming conventions help newcomers to interpret names (Section 8.1.3), naming issues remain a problem for newcomers (Appendix G.2.4).

- **Experts should describe the execution flow of highly delocalised code.** Newcomers find delocalised code (for example, as found in many event-driven systems) particularly difficult to follow at first (Section 8.4.5). Diagrams of typical execution flow are helpful in this situation. Although debugger features are not usually used in sessions, using them to trace the flow of highly delocalised code may be an exception. (Section 8.4.4)

- **Experts should give reasons for design and implementation choices.** This information is hard to obtain, but knowing these reasons allows newcomers to avoid repeating past mistakes. However, unless reasons are written down, they are unrecoverable from the project materials (Section 8.1.3). Newcomers value the Rationale view of code (Appendix G.2.1).

### 9.6 Evaluation of recommendations for practitioners

The recommendations described in this chapter are aimed at addressing the issues seen or reported by participants, and in many cases they are based on practices seen in some but not all organisations, or on recommendations made by participants. Despite this grounding in ecologically valid data, it is possible that the recommendations would not alleviate those issues in practice. A full evaluation is left for future work, but this section outlines some approaches for investigating the effects of the recommended changes.
9.6.1 Evaluating individual recommendations

One approach to the evaluation is to test recommendations individually, measuring their effect on the issues they are intended to address.

Many of the recommendations could be converted into hypotheses and tested using quantitative methods. For example, the first recommendation is that the expert should drive until the newcomer has grasped the high level overview of the project, and from then on the newcomer can drive. This is intended to improve the newcomer’s ability to locate code elements without back-seat driving by the expert. This could be tested in a laboratory setting: first, the newcomer is introduced to the codebase in one of three ways (expert driving, newcomer driving or expert followed by newcomer), and then the newcomer is tested on their ability to locate code elements. Since newcomers may retain information differently depending on its intended purpose (Pennington, 1987), the newcomer should be told that the session will be followed by the development task discussed in the session, but there is no need for the newcomer to actually perform this task.

The advantage of this approach is that the codebase, tools, introduction content and other variables can be controlled so that results can be compared across tests. However, wider issues of participant motivation and task realism are difficult to eradicate. The participants must be representative of the target group (newcomers, not novices or students) which can be costly to arrange. More importantly, it is difficult to be sure that laboratory results will generalise to software development practice. While an individual recommendation might have the desired effect, there may be other factors at work that influence its adoption and efficacy, and a laboratory setting is unlikely to uncover these issues.

Other recommendations are not so amenable to experimental investigation. For example, several recommendations are geared towards ensuring the newcomer has their environment set up ahead of their need for it, to improve newcomer retention of session content. This could be evaluated through a more focused field study that follows newcomers for their first month. Newcomers would be asked to record the development environment they are given on arrival (if any), the help they are given (through sessions, documentation, automation and so on) and the issues they encounter (documentation out of date, missing licences, gotchas and
so on). By collecting experiences from a range of companies, the relationship between expert help and newcomer problems with their environments could be evaluated more closely to understand which actions are most useful in reducing the time until the newcomer achieves full fluency.

While participant self-recording can work (e.g. Berlin and Jeffries, 1992), data collection may be affected by time pressures, social issues or empirical effects. However, this approach has the advantage of ecological validity.

9.6.2 Evaluating recommendations as a whole

An alternative approach is to evaluate the recommendations as a whole, by asking organisations to adopt them for genuine onboarding sessions. Rather than focus on the effect of individual recommendations, this allows evaluation of the practicality of the proposals and of interactions between recommendations. This qualitative technique would not quantify the effects of the changes; instead, post-session interviews with the participants would identify whether the recommendations were having the desired results.

Since many of the recommendations are drawn from session observations, it would be unsurprising if one or more were already standard practice for a given participant. Therefore, one of the first tasks is to establish which recommendations would constitute changes to practice. Additionally, some of the remaining recommendations might not be applicable to the participants’ context (for example, where participants use very different environments).

Having identified applicable changes, it is possible that participants might not use them in the session. This in itself is an interesting result, as it may indicate that a recommendation is difficult or awkward to implement. Session observations would check the extent to which recommendations are used in the session, rather than relying on participant reports, as participants might not report ‘non-compliance’ accurately.

The outcome of a successful onboarding session (or an entire onboarding process) is difficult to measure, either quantitatively or qualitatively. Here, post-session practitioner interviews are used to learn
whether changes were practical and useful. Both the expert and newcomer should be interviewed individually, because many of the changes are targeted at altering the tradeoffs between the two, and it is likely that their opinions of these changes will differ.

A suggested case study outline is as follows:

1. Obtain consent from participants.

2. Have participants fill in a questionnaire to establish their background as developers.

3. Meet the expert and discuss the recommendations and the expected effects. For each recommendation, note whether this is already standard practice, and if not, whether the expert is willing to trial the changes in the next session. Leave a copy of these notes with the expert as a reminder.

4. Observe the session without interfering. For each recommendation, note whether it was used in the session. Note also any difficulties that the participants have during the session.

5. After the session, interview the expert. For each recommendation, ask the expert:
   - For recommendations not used, what caused this decision.
   - What effect using (or not using) the recommendation had on them, on the newcomer and on the session as a whole.
   - Whether the expert believes the recommendation is valuable, and under what conditions.

6. After the newcomer has attempted to complete a development task, interview them to find out, for each recommendation:
   - What effect using (or not using) the recommendation had on them, the expert and the session as a whole.
   - What effect using (or not using) the recommendation had on their ability to complete the task.
   - Whether the newcomer believes the recommendation is valuable, and under what conditions.
CHAPTER 9. RECOMMENDATIONS FOR PRACTITIONERS

Analysis of these sessions would allow evaluation of each recommendation in several dimensions:

**Efficacy** - whether the change address its corresponding issue(s) as predicted.

**Conditions** - when the recommendation does and does not apply.

**Conflict** - whether the expert and newcomer have different opinions on the recommendation’s value.

**Practicality** - problems that make the recommendation hard to adopt.

### 9.7 Evaluating Applicability of the Theory

Applicability of a theory is judged by asking “Do the findings offer new explanations or insights? Can they be used to develop policy, change practice, and add to the knowledge base of a profession?” (Corbin and Strauss, 2008). Having developed a set of recommendations based on the theory, its applicability can now be evaluated.

#### 9.7.1 New explanations and insights

The theory provides structure and vocabulary for describing the content and configuration of onboarding sessions, and highlights which of these session activities are most valuable to newcomers.

#### 9.7.2 Use in practice

The theory identifies common problems that arise during and after sessions, affecting both newcomers and experts. These were described in Section 8.4, and the evaluation of fit indicates that the problems are encountered by practitioners worldwide. In this chapter, the problems were addressed through a set of suggestions, which could readily be applied by practitioners. Many of the suggestions are derived from practices seen in a minority of cases, indicating that they are highly workable but that these ‘best practices’ are not yet widespread amongst
practitioners. However, it is left to future work to evaluate the effects of the recommendations.
Chapter 10

Discussion

In accordance with the Grounded Theory methodology, the findings of this study were derived from the data without direct influence from other theories and findings in the literature. In this chapter, the findings are compared to the literature for contextualisation, support and discussion of new and different results.

10.1 Onboarding Sessions and Program Comprehension

Program comprehension is defined by Boehm-Davis et al. (1996) as “reconstructing the logic, structure and goals that were used to write a computer program”. For the newcomer, these are important targets for onboarding sessions, but just as software development incorporates more than just programming, so onboarding sessions are about more than program comprehension. Experts represent the logic (Algorithmic view), structure (Structural view) and goals (Rationale view), but the Rationale view includes more than ‘goals’ and experts also offer a Temporal view reflecting the realities of long-term software maintenance. The observed sessions included many of the components of program comprehension identified in the literature review, but also further activities that reflect the realities of working on an ever-changing, multi-developer project.
10.1.1 Reflections on program comprehension literature

The program comprehension literature describes beacons as sets of features in program code that indicate certain structures or operations, and experienced programmers pay close attention to them in order to verify and discover hypotheses (Section 2.2.1). In sessions, experts draw the newcomer’s attention to important lines of code (Section 8.1.2) and explain why they are important. This supports the continued importance of beacons. However, many newcomers report that this level of detail is not required in sessions because, as Brooks described, experienced developers are able to extract this information from the code themselves. Despite this, many newcomers take advantage of the expert’s knowledge to confirm their hypotheses about the code by asking questions or stating hypotheses (Section 8.1.2).

Languages and programming paradigms have developed since beacons were first proposed and investigated. Future work might examine expert explanations for beacon characteristics in current languages.

Plans (Section 2.2.2) are standardised structures that developers recognise even in unfamiliar code. In sessions, experts group lines of code by their function at varying levels of abstraction, and refer to them by name with varying levels of standardisation (Section 8.1.1). These observations support the use of plans in expert explanations, and provide examples of the names given to plans by developers. However, they also indicate alternative grouping strategies, such as ‘boilerplate’ (template) code being grouped separately to ‘meat’ (code considered important).

Comprehension strategies can be divided into ‘top-down’ and ‘bottom-up’ (Section 2.2.4). Experts were occasionally observed using bottom-up strategies to reconstruct the meaning of code, but this was unusual and was associated with a lack of high-level knowledge of the code. More often, experts provided a top-down view of code, starting with the high-level structure of the codebase (Section 8.1.1) before going into more detail. Newcomers strongly supported the top-down approach, stating that the high-level view was valuable session content, while low-level information was of very little value because newcomers can reconstruct it when required (as demonstrated experimentally).

Work on integrated models of comprehension (Section 2.2.5) linked a
programmer’s prior knowledge of the domain to their strategies for understanding the code. This explains newcomers’ enthusiasm for high-level information about the codebase (a top-down or domain model) because it makes more comprehension strategies available for use. In particular, in the top-down comprehension process as described by von Mayrhauser et al. (1997), a developer with prior knowledge of the code or domain can build hypotheses about the code and check them against the source code. With an expert present, the expert can provide domain knowledge (Section 8.1.1) and link it directly to code (Section 8.1.2), or quickly confirm or deny the hypotheses that newcomers create (Section 8.1.2). Experts were even observed guiding newcomers away from plausible but incorrect hypotheses. These actions provide the newcomer with shortcuts to a top-level model, even if the newcomer is not familiar with the domain. This information is therefore regarded as highly valuable.

Experts also provide shortcuts to the situation model, which would normally be based on domain knowledge and/or a program model. Industry-standard names for common abstractions (Section 8.1.1) allow newcomers to interpret situation-model information without knowledge of low-level code.

The program model is built from the code itself, and unlike the other two models does not take input from either domain knowledge or the other models. Experts are not able to provide significant shortcuts to the program model in the sessions, and discussion of code at this low level is not regarded as valuable by newcomers; as competent developers they are able to build this model independently as required, and the presence of another developer may even be a distraction to this mental process.

10.1.2 Additions to Program Comprehension

The Structural and Algorithmic views of code presented by experts have clear parallels in the software comprehension literature, but the Rationale view is less represented and the Temporal view is rarely exposed by program comprehension studies.
CHAPTER 10. DISCUSSION

The Rationale View

Von Mayrhauser et al. (1997) were surprised to find that the developers in their industrial studies voiced very few ‘why’ hypotheses (concerning the purpose of a function or a design choice) as opposed to ‘what’ and ‘how’ hypotheses, and conjectured that the inability to form ‘why’ hypotheses was due to a lack of prior experience with the system. Similarly, Ratanotayanon and Sim (2006) found newcomers asking very few, if any, ‘why’ questions of the ‘experts’ in their studies. In this experiment, the ‘experts’ had little expertise, having only 90 minutes’ experience on the system.

However, in these sessions, experts provided a Rationale view of code including the reasons for some design decisions, and some newcomers asked ‘why’ questions about design decisions, particularly those that did not immediately make sense (Section 8.1.3). This is mirrored by Ko et al.’s (2007) observation that developers sought rationale information from knowledgeable co-located co-workers, and Sharif’s (2012) report of open-source developers asking for rationale information through mailing lists (‘Why’ questions and ‘System Design’ topics). In his study, ‘Why’ questions caused the least discussion and the longest answers, suggesting that only certain developers had the knowledge required to answer these questions.

A possible explanation for the discrepancy with Sharif’s (2012) study is that developers’ questions (and voiced hypotheses) are matched to the perceived capabilities of the information sources. While algorithm questions can always be answered through detailed study of the code, rationale questions can often only be answered by a system expert; the newcomer could develop different, reasonable hypotheses about an implementation choice, but the information required to accept a hypothesis may not be present in the code. Developers know that experts (especially original authors) are the most effective source of rationale information, and may not attempt to seek answers on that topic from information sources (such as a non-expert guide or only the code) that cannot provide it. Thus, because von Mayrhauser et al.’s (1997) model was based on programmers’ actions with limited information sources, it could not incorporate this aspect of comprehension.

In contrast, Naur (1985) proposed a view of programming as theory building, i.e. “The programmer having the theory of the program can
explain why each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort.” In this view, programs ‘die’ (that is, can no longer be maintained) when there is nobody left who has the theory, and ‘revival’ (building a new theory) is difficult and likely to result in a partially incorrect theory. Examining the Rationale view issue from this angle, onboarding sessions and other interactions with experts (unlike unguided program comprehension) allow the theory of the program to be passed from expert to newcomer, prolonging the life of the program.

The Temporal View

Like the Rationale view, the Temporal view (Section 8.1.4) is not represented in program comprehension literature because of the limited circumstances upon which the comprehension model is based. The majority of studies required comprehension or a once-off modification of a static code artifact. In that scenario, there is no motivation for the newcomer to consider temporal aspects, no source of temporal information and, where the code was written specially for an experiment, there are no artifacts of temporal activity (such as changes in program conventions) to trigger Temporal view considerations. In contrast, in a genuine onboarding situation, temporal considerations become a part of the program comprehension process, providing an orthogonal view that supplements the ‘snapshot’ of the code as seen in the session (or in most studies). This concept also appears in Murray and Lethbridge’s (2005a) study of experts’ descriptions of their systems.

10.2 Onboarding Sessions and Information Seeking

As might be expected from the information seeking literature, software developers were observed using and discussing a variety of information sources, including the source code, documentation and other developers (Section 2.3.2). Other sources included reference materials for third-party libraries, and code artifacts such as log files.
Other than the newcomer asking questions of the expert, information seeking activity was rare in onboarding sessions, since the expert is knowledgeable (experts occasionally sought information from co-workers or environment setup documentation). More commonly, experts use the session to shape the newcomer’s information seeking strategy. This provides newcomers with shortcuts to a source’s trustworthiness (Hertzum, 2002) and cost to use (Seaman, 2002).

Developers’ specific information needs depend on their development tasks, but common difficulties for newcomers are described in Section 8.4.5. In particular, newcomers without working development environments have difficulty accessing a crucial information source, the code itself, and their information seeking is impaired by their lack of project terminology. This suggests the existence of a set of “information seeking prerequisites” that should be considered along with sources and needs in discussions of information seeking.

No participants were observed using tool support for information seeking during the sessions (although the basic IDE navigation and search capabilities were sometimes used to jump to known locations). However, this may be because the expert or a co-worker could generally answer the newcomer’s questions without seeking information from artifacts.

10.3 Onboarding Sessions and Concept and Feature Location

Onboarding sessions include matching code to its responsibilities and vice versa (Section 8.1.2). However, this concept and feature location is performed by the expert from their knowledge of the code, and no techniques or tools (beyond searching to jump to known code locations) were used in the sessions.

Newcomers do not mention using concept or feature location tools, but report that the high-level view of code is valuable for helping them predict where code is located.
10.4 Onboarding Sessions and Push and Pull

If onboarding sessions are judged by the criteria of Cybenko and Brewington (1999), their content is almost entirely information push, with rare examples of information pull when a newcomer’s question is answered. Thus, the content is determined largely by the expert, and this is a result of the expert taking the director role (Section 8.4.1) which was the most common session configuration and generally considered to be the most effective. The strong role of information push makes this study unusual.

Using the more detailed criteria developed in Section 2.5 from Deken et al. (2012), the majority of the interaction is active expert push. Notably, when experts shape the newcomer’s information seeking strategy, there is a bias toward active sources (i.e. co-workers) rather than passive sources (e.g. documentation).

The results of this study support many of the findings of Murray and Lethbridge’s (2005a) push-based investigation. Although the setting was artificial and diagram based, many of the cognitive patterns are recognisable in the onboarding sessions, indicating that they generalise to natural onboarding sessions and are not confined to whiteboard sessions. For example, the ‘Big Picture’ pattern is supported by the widespread use of abstraction and a high-level, top-down view of the system.

Most of the patterns are not presented in enough detail for an in-depth comparison with the findings. However, snapshots and temporal details were described more fully by Murray (2006). Snapshots are defined there as “an interval or moment in the development of a software model when the model is cohesive or conceptually whole”. Snapshot identification is useful because snapshots mark a moment where insight about a system can be achieved.

In Murray’s (2006) snapshot study, every participant began by producing an ‘infrastructure’ snapshot, and used this as a base for the remainder of the session. Infrastructure snapshots are characterised as a general overview of fundamental and architectural details. This is mirrored in onboarding by the experts’ top-down explanations beginning with the overall architecture, and the newcomers’ responses indicate that this is an effective starting point. Unlike the snapshot results, not all the onboarding experts began with this top-down view. In some cases this was because the newcomers already had this knowledge, but in others it was
associated with a lack of knowledge on the expert’s part. A developer’s ability to produce an infrastructure snapshot is perhaps an indicator of their ability to explain the system to a newcomer.

‘Weak snapshots’ were produced when their participants lacked expertise on an area. Beginning with a self-evaluation of their knowledge, the participant went on to provide a general discussion instead of the meaningful structural and/or functional descriptions of the other snapshot types. Corresponding activity was seen in the sessions of this study, when experts reached the limits of their knowledge of the system. In onboarding terms, the experts’ self-evaluation was useful to newcomers for shaping their information seeking strategy, but the general discussion, where the expert gave vague and sometimes guessed content, was poorly received.

The snapshot study did not encounter any snapshots without drawings, but this is not surprising given its focus on whiteboard diagrams. The onboarding observations suggest that snapshots do occur without drawings and can be based on lines of code, software demonstrations and IDE representations of structure. A full analysis of this is left for future work.

Murray (2006) also expands the Temporal Details pattern, in which key historical moments in the evolution of the model of the system (and thus in the system itself) are captured by snapshots. This pattern addresses the issue that the ‘final’ model is needed to work effectively on the system, but this model may not include important historical decisions and their rationales, making its form difficult to comprehend.

This is clearly related to the Temporal and Rationale views described in this onboarding study, but with some key differences. Murray’s (2006) Temporal Pattern describes a series of past states of the system, each one a ‘before’ or ‘after’ state of an important change, ending with the final model. In contrast, the Temporal views provided by experts during the onboarding sessions include past, current and future changes in acknowledgement of the ongoing maintenance of the system. While completed changes can be viewed as events, changes are (often ongoing) processes that reflect evolving design philosophies and code conventions. The Rationale view of the code, in particular the storytelling aspect, hints at alternate states (‘before’ and ‘after’, ‘now’ and ‘later’, or ‘implemented’ and ‘alternative’) but the focus is on the rationale for moving between the states.

While they differ in their representations, both Murray’s (2006)
Temporal Details pattern and the Temporal and Rationale views confirm Naur’s (1985) view that this is crucial information for the long-term maintenance of a program. Its near absence from the program comprehension literature is due to a focus on code without temporal context and on participants without temporal or rationale knowledge of code.

10.5 Onboarding Sessions and Onboarding Literature

Sim and Holt’s (1998) study of ‘software immigrants’ is one of the closest to this research in terms of context. Their seven onboarding patterns can readily be related to the findings.

Their first pattern (Sim and Holt, 1998) is that mentoring is effective but inefficient. Effectiveness and inefficiency are difficult to measure quantitatively in this setting, but in this study, many of the observed sessions included content that was of no use to the newcomer (at least before the interview) because it was unrelated to their task or because newcomers forgot the session content before they had a chance to apply it; avoiding this would improve efficiency. Given the tradeoff between optimising sessions for the expert and for the newcomer, it is also likely that an expert-newcomer pair will have differing opinions of the sessions’ effectiveness and efficiency.

Their second pattern is that the lack of documentation forces newcomers to rely on experts for information. There is some truth to this, but information seeking strategies vary between teams and some encourage the use of co-workers as information sources even when there is a strong culture of documentation. At the far end of the scale, newcomers can be faced with so much documentation that they need expert guidance to navigate it.

Thirdly, newcomers were frustrated by administrative and environmental issues. Their third pattern is strongly supported by Section 8.4.3. Sim and Holt speculate as to the cause of this feeling of frustration; the newcomers in this study report that it is because the lack of a working development environment prevents them from making
progress, not just through contributions but also with their comprehension of the code. The authors recommend automation of environment setup, but the experiences of the newcomers in this study show that this is not necessarily a panacea because there may still be difficulty accessing the automation system.

Pattern four describes the first tasks given to newcomers. In contrast to the first tasks in this study (Section 5.2.4), those in Sim and Holt’s (1998) study were all given shortly after their newcomer had established a working environment; however, Section 8.4.2 reveals that there can be a wide variation in this time which affects the value of the onboarding session.

Their fifth pattern states that mentors tend to pass on low-level information about the system, and the sixth that programmers who preferred a bottom-up comprehension approach had a smoother naturalisation. The findings of this study, coupled with the discussions on program comprehension models, suggest that these two patterns are related: because their mentors passed on low-level instead of high-level information, their newcomers were limited to a bottom-up comprehension approach. Had their mentors provided a high-level view of the system, their newcomers would have had a wider range of comprehension strategies available to them. The evaluation results in Chapter 7 showed that newcomers strongly prefer high-level information from onboarding sessions.

Their final pattern is that onboarding is more effective when newcomers have a personal interest in the system. This study did not reveal any concepts relating to this.

Overall, the depth of this study produces several important additions to Sim and Holt’s (1998) findings, while the differences illustrate the value of variation in the sources. This variation was not present to the same extent in their four participants from the same company.

Sim and Holt (1998) piloted the provision of a two-hour video course on high-level structure to newcomers, with moderate success. The findings in Chapter 9 suggest that, while high-level content is valuable, the course would need to be structured to avoid information overload (for example, by including breaks in which newcomers could explore the code independently). It also needs to be targeted to the newcomer’s task, so a selection of smaller video ‘modules’ might be more appropriate.
The study by Dagenais et al. (2010) provided a broad view of the social and technical aspects of the ‘project landscape’ that newcomers encountered. This study provides a deeper investigation of the ‘Initial Guidance’ part of their ‘Early Exploration’ aspect, while the ‘project landscape’ study provides broader context for this study (which focuses only on onboarding sessions and not the whole onboarding process). The two studies are in agreement on the positive effects of mentors and co-workers, and the negative effect of development environment difficulties. Using the landscape metaphor, the authors suggest that ‘guides’ (expert mentors), ‘paths and signposts’ (e.g. issue tracker histories) and ‘maps’ (routes between landscape features) help newcomers to find their way around a new project. Continuing this metaphor, Chapter 9 contributes advice for experts on effective guidance, and highlights the issue of missing ‘equipment’ for exploration in the form of prerequisites such as project terminology and a development environment.

The HP Labs study (Berlin and Jeffries, 1992; Berlin, 1993) confirms once again the issue of the development environment, and also describes the wider conflict between the newcomer’s need for information and desire to avoid inconveniencing the expert; Section 8.4.1 showed that this conflict continues even during onboarding sessions and affects their configuration and value. Berlin (1993) reports experts mentoring effectively without preparation or training, but the recommendations in Chapter 9 are largely aimed at the expert; expert contributions to the early stages of the onboarding process can be streamlined to provide more efficient assistance to newcomers.

10.6 Onboarding Sessions and Other Research

The category of ‘storytelling’, described in Section 8.1.3, illustrates experts passing on project history to newcomers to help them avoid known issues. Wende et al. (2009) also report on storytelling for tacit knowledge transfer between distributed teams with differing cultures.
10.7 Threats to validity

Section 4.3.1 describes some indications that participants were aware of the recording equipment, raising the possibility that they behaved differently under study conditions. However, the evaluation from developers who were not recorded does not indicate that this had a significant effect. Since onboarding sessions naturally include a minimum of two people who are paying attention to one another, it is likely that the presence of a researcher and/or recording equipment has little additional effect compared to introducing an observer to an individual’s activity.

One exception is that the non-recorded participants reported more variation in the owner role, whereas most recorded sessions used the expert’s machine. This may have been a side effect of the request to install screencapture software on the machine(s) to be used in the session. This could have been mitigated by arranging to install the software prior to the session throughout the study (this was only done in later sessions to reduce disruption). However, the recorded sessions still included enough variation in the owner role to demonstrate its connection to the other session roles.

Participants were recruited through convenience sampling, which raises the possibility that the participants had more confidence in their onboarding session techniques than the general population of developers. This represents a threat to validity, but the alternatives - having developers participate against their will or without knowledge of recording - are clearly unethical. The wide variation in session styles and expert preparation, and the results of the evaluation, suggest that the recordings were representative of onboarding sessions in industry.

10.8 Summary

Comparing this study to the literature reveals support for many of the major findings, in particular the issues commonly encountered by newcomers and the neglect of temporal and rationale information in program comprehension models.
Chapter 11

Conclusions

This research originated from two related issues: firstly, that understanding unfamiliar software is a difficult task for developers, and secondly that the program comprehension literature lacks the perspective of information push. Previous studies of onboarding have taken broad or longitudinal approaches with little attention paid to the content of onboarding sessions. However, these sessions are not only an important part of the onboarding process, they are also a natural source of actively pushed information towards program comprehension.

This exploratory study, adding to the growing number of GT studies in the field of software engineering, supplemented observations of twelve onboarding sessions with interviews and questionnaires for triangulation and evaluation. As a result, all five of the research questions have been addressed; the first four, focusing on description of onboarding sessions, in Chapter 8, and the last, addressing session problems, by Chapter 9.

11.1 Summary of Findings

This section summarises the responses to the five research questions. The first four are answered through a Grounded Theory of onboarding, and the fifth in the form of recommendations for practitioners.
11.1.1 How do experts represent the code to the newcomer?

The experts’ representation of the codebase can be described in terms of four views. The order of the views presented here does not correspond to the session structure; session structure varied greatly, with the exception that many sessions began with a high-level overview of the architecture.

The Structural view provides a high-level view of the components of the system, which is described through abstraction and the characteristics of the code. Abstract views of the code include a high-level view of the major components, use of design pattern names, and naming blocks of code. Code characteristics include simplicity and complexity, similarity and difference, the cost and size of code, and its importance overall or within the current context. Together, the structure and the characteristics help the newcomer to navigate the code.

The Algorithmic view involves tracing the flow of execution of a selected feature or concept through the code. Often, the selected feature or concept is first demonstrated or described, and the tracing can be at varying levels of detail. This view provides a mapping between the feature and its implementation.

The Rationale view covers the requirements and reasons that led to the system’s current design. Rather than provide requirements documents, experts demonstrated the system from an end-user’s point of view, and showed or sketched diagrams to illustrate non-obvious business rules. Sometimes the clients were described, to add context to the requirements. Reasons behind design choices included architectural patterns, project-specific conventions, workarounds and historical effects.

Finally, the Temporal view informs the newcomer about previous, current and future development work that impacts the code. This includes temporary fixes, changes to project conventions and explanations for other temporal artifacts that might otherwise confuse a newcomer. The Temporal view provides a crucial extra dimension to what would otherwise be a static snapshot of the codebase.
11.1.2 How do experts support newcomers in locating information?

While experts supply newcomers with information during the sessions, this is not a sustainable way for the newcomer to learn everything about the code. Thus, experts also use the sessions to shape the newcomer’s strategies for locating information themselves, supplementing the newcomer’s general strategies with project-specific guidance.

An important prerequisite for this is the project’s terminology; without this, newcomers struggle to phrase queries and interpret information.

Recommended information sources include project documentation, co-workers, reference materials, and code and code artifacts. In addition to describing where to find these materials, experts sometimes provide opinions on their qualities as information sources. This includes the currency of documentation, the expertise (and otherwise) of co-workers including the expert themselves, and the ease of use of reference materials.

The expert also provides strategies for information seeking, including guidance on which source is most likely to be beneficial, and when it is appropriate to ask co-workers for help.

11.1.3 How do experts support newcomers in making contributions?

The support provided by the expert depends on whether the newcomer has been given a task before the session. In addition to an overview of the code and an information seeking strategy, all newcomers can be provided with general advice for working on the project, such as interpretation of project conventions, prediction of common long-running tasks and workarounds for known issues. However, when the newcomer has a development task in mind, the expert can supply more targeted advice: pointing out relevant and irrelevant code, recommending design changes and providing test data.

11.1.4 What are the problems with onboarding sessions that reduce their value?

Both experts and newcomers experience difficulties during the sessions, and the characteristics of the sessions influence their effectiveness.
The session’s configuration can be viewed in terms of three roles, each of which affects the session. The driver role refers to the individual who has control of the keyboard and mouse: the expert driving can create a more fluid experience, but this comes at a cost to the newcomer’s level of engagement. The director is controlling the content of the session; as the newcomer gains knowledge of the system, this role becomes more appropriate for them. The owner is the normal user of the machine in the session, and this role is strongly associated with both the driver role and with the provision (or otherwise) of the newcomer’s development environment. Overall, the session configuration can be seen as a tradeoff between convenience for the expert and engagement for the newcomer.

The timing of the session is important because without the opportunity and motivation to work on the code soon after the session, the newcomer is likely to forget its content. When sessions do not include task discussion, they are less likely to be relevant to the newcomer’s eventual work.

Newcomers have difficulty making use of the session without a working development environment, and with environment setup due to the rights and licences required. Documentation and automation does not fully address this problem. Newcomers value expert help with this issue, even though experts occasionally find that they have forgotten the details needed by the newcomer.

While diagrams are not unusual in sessions, newcomers commonly request two types in particular: an abstract, high-level overview and a detailed view of the area relevant to their task. When these are not provided, some newcomers ask experts to provide these diagrams after the session, or use diagram generation tools. However, each diagram’s context and notation must be understood in order for it to be of any use, and too much detail can be as problematic as too little.

Experts do not maintain a complete knowledge of all the code. This presents both the expert and newcomer with difficulties in the session. Where the expert has forgotten details, or makes mistakes in the explanation, the session is harder to follow and less valuable for the newcomer. Some experts anticipate this difficulty and review or comment the code before the session.

The project’s terminology is also initially problematic for both newcomer and expert. Without this shared vocabulary, newcomers struggle to express
Most experts either created a plan for the session, or followed an important execution path through the code. The newcomer’s questions sometimes threatened to derail this plan, and experts had to decide whether to delay answering or allow tangents to their planned path through the code. Even with preparation, some experts felt under pressure to fill the session with as much information as possible, which had the potential to overwhelm newcomers.

In general, the difficulties for experts occur before or during the session - for example, when reviewing the code or when a newcomer asks difficult questions. For newcomers, difficulties occur during or after the session - for example, being slowed down by a lack of correct terminology, or forgetting information afterwards.

11.1.5 How can onboarding sessions be improved?

The recommendations made in Chapter 9 can be summarised as follows:

- Sessions should take place after the newcomer’s task has been assigned.
- No code should be explained in a session until the newcomer’s development environment has been set up.
- The expert should help the newcomer with development environment setup.
- At the very beginning, the expert should drive.
- The expert should use this time to provide a high-level overview of the system.
- When the newcomer has a high-level overview, they should take over the driving role from the expert.
- In general, the expert should take the director role during sessions.
- The session should always be conducted on the newcomer’s machine.
- Experts should review code prior to the session.
• Experts should provide diagrams of the high-level structure of the system early in the session.

• Experts should provide detailed diagrams of task-relevant structures.

• The session should include discussion of the newcomer’s development task.

• Experts should trust newcomers’ development skills.

• Experts should help newcomers to locate answers to questions that the expert cannot answer.

• Experts should keep sessions short.

• Newcomers should make notes and keep reference materials.

• Experts should define terminology before using it.

• Experts should provide solutions for ‘gotchas’.

• Experts should provide the meaning of unclear names.

• Experts should describe the execution flow of highly delocalised code.

• Experts should give reasons for design and implementation choices.

11.2 Novel contributions

The novel contributions of this study are threefold: the description of onboarding sessions, the push-based information to be incorporated into the pull-based view of program comprehension, and a set of recommendations for practitioners.

11.2.1 Description of onboarding sessions

While the literature includes several studies of the onboarding process, this is the first to examine early onboarding sessions with a focus on comprehension. This has resulted in a rich description of the content of onboarding sessions across a variety of organisations, platforms and nationalities. It identifies, names and describes the four views through
which experts describe their code. It describes the expert’s shaping of the newcomer’s information seeking strategy, strengthening the link between the information seeking literature and industrial practice, and shows the general and specific support that experts offer to newcomers.

In addition to descriptions drawn from observations of the sessions, interviews with newcomers revealed problems with the sessions that were not always apparent from the observations. Some of these problems have been known for some time, particularly the development environment issue and the problem of information overload, while others have not previously been described, for example the session timing issues and the effects of the three roles that describe a session’s configuration. By comparing a variety of sessions, this study contributes links between onboarding session practices and problems, showing where changes in practice could address those issues. The study also contributes reports of the issues faced by experts, showing that in contrast to previous reports, experts would benefit from guidance or training for their role in onboarding.

11.2.2 Contribution to the literature

In contrast to the pull-based model of program comprehension, in which only Structural and Algorithmic information is sought by newcomers, experts push four views of the code: Structural, Algorithmic, Temporal and Rationale. This is due to important differences in the context and available content of the studies that contributed to the model. By incorporating the latter views, our understanding of the program comprehension process can be developed from the limited model based on newcomers’ best efforts at a difficult task, to a broader model informed by and incorporating both newcomer actions and expert guidance.

11.2.3 Recommendations for practitioners

Chapter 9 contributes a set of recommendations for practitioners involved in onboarding sessions (both experts and newcomers). The recommendations are practical (with many of them drawn from observed behaviour in sessions) and backed by rationale drawn from comparisons of different session practices and resulting outcomes. Methods for evaluating
these recommendations are also provided for future use.

This is a first attempt at establishing best practice for onboarding sessions and an attempt to address the many problems and inefficiencies of the onboarding process. This is timely, as the developer community has also begun to discuss onboarding practices based on individual experiences (for example, MacIver, 2013; Mueller, 2012).

11.3 Future work

11.3.1 Evaluation of recommendations

While the theory of onboarding sessions has been evaluated, the recommendations derived from it have not. Section 9.6 describes some approaches to evaluation.

11.3.2 Temporal and rationale comprehension

This study, in common with Murray and Lethbridge (2005a), highlights the importance of temporal and rationale information in comprehending a system under active maintenance - an aspect that is not well represented in program comprehension models. More work is required to integrate these crucial views into the models.

In contrast to the original participants, comments from the second evaluation (Appendix G.2.1) indicated that the importance of the Temporal view may be related to factors such as team size and code ownership policies.

11.3.3 Longitudinal efficiency

This study captured onboarding sessions at a range of locations and investigated their short term effects (from the session to the newcomer’s first task), but did not attempt to capture or investigate the long-term effects of onboarding sessions. A longitudinal study might provide further insight on the session optimisation issue; whether it is more efficient in the long run to optimise sessions for the expert or for the newcomer. This
type of study would also demonstrate whether developer fluency (Zhou and Mockus, 2010, 2011) progresses in the same way in collocated teams.

11.3.4 Distributed onboarding sessions

All the participant sets in this study were colocated, so the issues of offshore software development (including language barriers and cultural differences) did not feature in the findings. However, the evaluation revealed that onboarding sessions do not necessarily rely on colocation, and can use screensharing technology to provide shared context. Incorporating this variation would strengthen the theory.

11.3.5 Prevalence of onboarding culture in industry

Section Section 7.2.2 reports on an interesting characteristic of the second evaluation group. In this small sample there is a correlation between the time since they took both the newcomer and expert roles. A larger study, covering several different organisations, would shed some light on whether this is due to team or organisation-wide onboarding cultures or simply an individual preference for collaborative working.

11.3.6 Experts forgetting code

While some experts prepared for sessions by reviewing the relevant code, others did not expect to have any difficulties recalling the details of their codebases. However, in the sessions, it was common for experts to misremember code, leading to on-the-fly comprehension.

This is both a temporal issue (where it is caused by other team members changing code) and a comprehension issue (given that experts have more difficulty recalling unplanned code (Soloway and Ehrlich, 1984)). Future work might investigate whether both issues are accounted for in models of developer degree-of-knowledge of code (e.g. Fritz et al., 2010) and whether such models might assist experts by recommending areas for review before an onboarding session.
11.3.7 Modern beacons and plans

Beacons (e.g. Brooks, 1983) and plans (Soloway and Ehrlich, 1984) are widely accepted components of program comprehension models. However, languages and programming paradigms have seen several decades of development since these concepts were first described, making the original observations difficult to relate to current codebases. Industrial data sets such as this one could be used to identify plan and beacon characteristics, such as their names and purposes according to practitioners, in current languages.
Bibliography


Brooks, F. (1975), The mythical man-month: essays on software engineering, Addison-Wesley.


URL: www.atlasti.com
BIBLIOGRAPHY


URL: http://blog.smartbear.com/sqc/how-to-inherit-somebody-elses-code/

URL: http://www.atlasti.com


Riley, J. (1996), Getting the most from your data, 2nd edn, Technical and Education Services Ltd.


URL: http://www.videolan.org/vlc/


Wende, E., Philip, T. and Dubberke, S. (2009), Storytelling - an instrument to bolster knowledge transfer in offshore software projects, in ‘The Third Global Sourcing Workshop’.


212

Appendices
Appendix A

Information Sheet

The following two pages comprise the information sheet developed to provide potential participants with a quick summary of the research. The contents of the sheet evolved in response to frequently asked questions; this is the final version.
The Issue

- On joining a company, or moving to a new project, a software developer is faced with unfamiliar code and has difficulty contributing. Ramping up to full productivity can take up to six months.
- In research at Microsoft, 56% of developers reported that understanding someone else's code was a serious problem.
- Expert software developers are able to mentor new team members – but the experts aren't always available, and mentoring can be time-consuming.

Research Proposal

- To address this issue, I'm investigating how experts explain their code to new team members. Through video recordings, screen capture, code analysis and interviews, I'm seeking to understand what information about a codebase is most useful in this situation.
- This is a new approach to the problem. To ensure valid and useful results, it's important to involve industrial partners.
- More information about the data collection can be found on the reverse.

Outcomes

- This kind of research can answer questions such as:
  - Which parts of the code are important for a new team member to know?
  - How are diagrams used to convey information?
  - How useful was the explanation to the newcomer?
  - How could the ramp-up be improved?
- I hope that this research will lead to recommendations for mentoring techniques or tool support for new team members, reducing ramp-up time and increasing developer productivity.

If you have any further questions, or would like to participate, please contact me: (rebecca.yates@lero.ie). Thank you for your time.

Rebecca Yates is a PhD student at the University of Limerick, Ireland, supervised by Jim Buckley and Norah Power. Prior to this, she worked as a software engineer.
Thank you for considering helping with this research into expert explanations of software. This page details the steps typically required for participation.

**Beforehand:**

We arrange a time for the data collection session.

I am most interested in a newcomer's first introduction to the codebase. The walkthrough typically takes between one hour and half a day. Newcomers may be new hires, or existing employees moving to a different project within the company.

We set up an NDA if required.

This can be supplied by your company or created by the University of Limerick.

**On the day:**

The participants read and sign the Participants Rights document.

This document covers their rights regarding anonymity, their right to pause or abandon the recording for any reason, and describes how the collected materials will be used.

We install screencapture software on their machine.

E.g. the freely available CamStudio (unless a suitable program is already installed). This is because the text on screen doesn't show up clearly on the camcorder.

I place the camcorder and start recording.

This is a small camcorder on a desktop tripod, and is pointed at the screen. It's unobtrusive, and can be moved easily to follow the conversation to whiteboards or other locations.

The participants work as usual, with the expert explaining the code to the newcomer.

I move the camcorder if required, and don't interfere with the work in progress.

After the session, I take a photo of any diagrams that were used.

Like the screencapture, this just ensures a clear copy in case it's not readable on the video.

I analyse the source code under discussion.

This is for the quantitative part of the research, comparing the path of the explanation to the code metrics. All copied materials are stored encrypted using ecryptfs and are not released; if code cannot be removed, on-site access would suffice.

**Follow up:**

The participants complete an online questionnaire.

This is a short (~10 minutes) questionnaire on their background as software developers.

The newcomer gives a short interview.

This takes place around three weeks later, and asks how useful the explanation was. This can be completed in person or by phone and should take less than half an hour.

 Optionally, I can provide a report or presentation on this research.

To respect the privacy of the participants, this is a report on the research in general, and does not name any individuals or companies. As detailed on the previous page, this report would focus on parts of the code base that are important to new team members, how diagrams can be used effectively, the usefulness of different types of information and general improvements to the ramp up process.

If you have any further questions, or would like to participate, please contact me: (rebecca.yates@lero.ie). Thank you for your time.
Appendix B

Participant Background Questionnaire

The following five pages comprise the questionnaire that participants were asked to fill in after the onboarding session(s) and before the interview. The questions were presented online on SurveyMonkey.com and completed in the participants’ own time. As discussed in Section 4.3, the questions were iteratively developed as the study progressed, and the questionnaire presented here is the final version used.
Thank you for your interest in my research.

This questionnaire asks you for some information on your background and your software engineering experience.
2. Your background

<table>
<thead>
<tr>
<th>*1. Your name:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

2. How much commercial software development experience do you have?

<table>
<thead>
<tr>
<th>Years</th>
<th>Months</th>
<th>Number of different companies I worked at</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. How much other software development experience do you have? (e.g. voluntary open source contributions, coursework, personal projects)

<table>
<thead>
<tr>
<th>Years</th>
<th>Months</th>
<th>Number of different projects worked on:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. What is your highest qualification?

<table>
<thead>
<tr>
<th>Level</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Please list the programming languages that you have used.

**Languages I am currently very familiar with:**

<table>
<thead>
<tr>
<th>Languages</th>
<th>I used extensively in the past</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Languages I have used a little:**

<table>
<thead>
<tr>
<th>Languages</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Learning and mentoring

1. Thinking back to the last time you needed to understand unfamiliar code: what did you do to learn about the code and was it useful?

2. Thinking back to the last time you needed to explain unfamiliar code to someone else: what preparation did you do and what strategies did you use?
### 4. Your current project

#### 1. What is your current position in terms of this project?

#### 2. How long have you worked on this project?

#### 3. How familiar are you with:

<table>
<thead>
<tr>
<th>Item</th>
<th>I don't know what it is</th>
<th>I have never used it</th>
<th>I have used it a little / used something similar</th>
<th>I have used it a lot</th>
<th>I am an expert (others ask for my help with it)</th>
</tr>
</thead>
<tbody>
<tr>
<td>the programming languages used by this team</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the team's codebase in general</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the code that was under discussion during the recording</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the source control system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the issue tracking system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the domain you are working in (e.g. healthcare, accounting)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the team's products as an end user</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>design patterns (e.g. Observer, Strategy, Factory)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mentoring other developers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thank you for completing the questionnaire.

My research depends on your help. By allowing me to record your discussions, you'll be allowing me to analyse the way we learn about unfamiliar systems. I hope you will also find the results interesting.

If you have any questions, please contact Rebecca Yates (rebecca.yates@lero.ie)
Appendix C

Followup Interview

The following questions comprise the set of standard questions used in the followup interviews of newcomers. Like the background questionnaire, the standard question set evolved in response to the direction of the analysis, and the version presented here is the final version. The interviews took a semi-structured format, so additional questions were introduced in response to the participant’s answers and the unusual features of each session.

1. Please give the overall purpose of the software that was being discussed in the session.

2. Give an overview of what you discussed in that session.

3. When did you first see the code?

4. When did you start modifying the code?

5. Please highlight some things you learned in the session that proved useful when you were modifying the code.

6. With hindsight, what extra session content would have been useful?

7. How else could the have improved the session for you?

8. If you had to explain this code to another developer joining the project, what would you do?

9. [After explaining the concept of ‘the driver’] Do you think it is better for the expert or the newcomer to drive? Why?
10. How does the relative experience of the newcomer and expert affect the session? Why?

11. [At the end of the interview] Do you have any other comments about anything we’ve discussed?
Appendix D

Testing Fit Questionnaire

The following pages comprise the questionnaire that participants were asked to fill in after the interpretation of all the sessions. The questions were presented online on SurveyMonkey.com and completed in the participants’ own time. The development of this questionnaire is discussed in Section 7.1.
Introduction

This questionnaire is part of a study on onboarding in software engineering. This research aims to understand how newcomers are introduced to the codebase, and to suggest improvements to the process.

This online questionnaire should take no more than 20 minutes to complete. Your questionnaire answers will not be shared with anyone outside the research group. Your responses will be used in the evaluation of this research, and text responses will be anonymously quoted.

You and the data you provide will be anonymized in all reports using this data.

You have the right not to answer questions and to withdraw at any time. By continuing with this questionnaire you are deemed to have given consent to participate. If you have any questions, please contact the researcher. If you have any concerns, please contact the Chair of the Science & Engineering Research Ethics Committee at the University of Limerick.

Contact information:
Researcher: Rebecca Yates (rebecca.yates@lero.ie)
Supervisors: Jim Buckley (jim.buckley@ul.ie) and Norah Power (norah.power@ul.ie)
Chair of S&E REC: Dr Thomas Waldman (+353 61 202802)
Ref: 2013_07_26_S&E

About you

1. Your name:

2. How much commercial software development experience do you have? (in years)

3. How much other software development experience (e.g. personal projects) do you have? (in years)

4. How recently have you explained a codebase to another developer? (in days, months or years)

5. How recently has a developer explained a codebase to you? (in days, months or years)

Instructions

In this questionnaire, “expert” refers to an experienced developer who knows their codebase well, and “newcomer” refers to an experienced developer (not a novice) who does not know that codebase. The questions refer to sessions in which an expert explains the code to a newcomer, so that the newcomer can begin development work.

Each multiple-choice in this questionnaire is followed by a comment box. Comments are optional, but please expand on your choice if you feel strongly about the issue, or if the available choices don't adequately cover your response.

Helpful actions in sessions
6. What actions by the expert are useful or useless to the newcomer?

<table>
<thead>
<tr>
<th>Action</th>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>The expert goes through complex code line-by-line.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert describes the high-level architecture.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert describes the clients.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert points out simple and complex areas of code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert points out costs (e.g. in terms of memory use, speed or file size).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert names chunks of code (e.g. &quot;the initialisation stuff&quot;).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert goes through all the code line-by-line.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert points out similarities or commonly used idioms in the code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The expert points out the design patterns in use in the code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choices:

Choice of driver

7. In the session, who should drive (i.e. have control of the keyboard and mouse)?

- The expert should drive.
- The newcomer should drive (guided by the expert).
- The expert should drive initially and provide a high level overview, and then hand over to the newcomer.
- It depends on personal preference of both expert and newcomer.
- It doesn't matter.
- Other (please specify)
8. If the newcomer drives, they are more engaged because they cannot "switch off".

9. If the newcomer drives, the session is slower and less fluid.

10. If the newcomer drives, this improves their ability to find code independently after the session.

11. If the newcomer drives, it is hard for the expert to direct them (the expert may be "back seat driving").

12. If the newcomer drives, the expert can judge how well the newcomer is learning the system.

13. If the newcomer drives, it is hard for the expert to navigate and locate code.

Choice of driver

Expert: an experienced developer who knows their codebase well.  
Newcomer: an experienced developer who does not know that codebase.  
Session: a time when the expert explains their code to the newcomer.  
Driving: having control of the keyboard and mouse.

Please expand on your choice:

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
Driving: having control of the keyboard and mouse.

14. If the newcomer drives, the expert is prevented from going too quickly through the system.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

15. If the newcomer drives, they will find the session more confusing.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

### Ongoing changes to code

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

16. If the expert describes longer-term ongoing changes to the codebase, the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

17. If the expert points out temporary fixes in the code, the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

18. If the expert points out dead code, the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

19. Project conventions may change over time, resulting in two or more conventions coexisting in the codebase. If the expert discusses this, the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:
### Choice of machine

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

#### 20. Whose machine should be used in the session?

- [ ] The expert's machine.
- [ ] The newcomer's machine.
- [ ] It doesn't matter.
- [ ] A third machine.
- [ ] Both the expert's and newcomer's machines (assuming they can be placed side by side)
- [ ] Other (please specify) [ ]

#### 21. If the session uses the newcomer's machine, it will be interrupted by missing configurations or data.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice: [ ]

#### 22. If the session uses the newcomer's machine, the newcomer will not be blocked afterwards by environment problems.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice: [ ]

#### 23. If the session uses the newcomer's machine, the explanation is not as smooth.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice: [ ]

### The expert’s point of view

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
24. Without a thorough understanding of (part of) the code, the expert cannot present a high level view of it, and instead resorts to working through code line by line.

25. If the code is well commented, the expert will find this:

26. Experts have become so at home in the system that they forget to mention ‘gotchas’ - quirks of the codebase or the environment with known but non-obvious workarounds.

---

The expert’s point of view

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

27. Experts find the debugger a useful tool for explanations.

28. Experts remember how to set up the newcomer’s development environment.

---

Helpful actions in sessions

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
29. If the expert points out possible misconceptions (e.g. "it gets the information from the server, it doesn't do it locally"), the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

30. If the expert ends each chunk of explanation (e.g. "that's all there is to it") the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

31. If the expert uses the debugger to step through code, the newcomer will find this:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

### Ideal session characteristics

**Expert:** an experienced developer who knows their codebase well.

**Newcomer:** an experienced developer who does not know that codebase.

**Session:** a time when the expert explains their code to the newcomer.

#### 32. From the expert's point of view, the ideal session provides the newcomer with as much information as possible.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

#### 33. From the expert's point of view, the ideal session allows the expert to return quickly to normal work tasks.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

#### 34. From the expert's point of view, the ideal session does not waste time on environment setup.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:
Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

35. From the newcomer's point of view, the ideal session engages the newcomer, even at the cost of frustration to the expert.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
</table>

Please expand on your choice:

36. From the newcomer's point of view, the ideal session directly helps them to make early contributions.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
</table>

Please expand on your choice:

37. From the newcomer's point of view, the ideal session will finish early if the newcomer has enough information to be going on with.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
</table>

Please expand on your choice:

38. From the newcomer's point of view, the ideal session includes help with environment setup.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
</table>

Please expand on your choice:

39. From the newcomer's point of view, the ideal session gets them past problems that would otherwise block them.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
</table>

Please expand on your choice:

Ideal session characteristics

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

Seeking information

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
### 40. What information can the newcomer find without expert help?

<table>
<thead>
<tr>
<th>Information</th>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to use the editor or IDE.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The reasons for implementation choices.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to run the software as an end user.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The location of test data.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether documentation is up to date.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether this is the latest version of the source code.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Workarounds for common problems.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to set up the development environment.</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Please expand on your choices:

---

### Seeking information

**Expert:** an experienced developer who knows their codebase well.  
**Newcomer:** an experienced developer who does not know that codebase.  
**Session:** a time when the expert explains their code to the newcomer.

### 41. If the expert gives the limits of their own knowledge, the newcomer will find this:

<table>
<thead>
<tr>
<th>Information</th>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to use the editor or IDE.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The reasons for implementation choices.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to run the software as an end user.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The location of test data.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether documentation is up to date.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether this is the latest version of the source code.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Workarounds for common problems.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to set up the development environment.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Please expand on your choice:

---

### 42. If the expert gives their opinion on the qualities of information sources, the newcomer will find this:

<table>
<thead>
<tr>
<th>Information</th>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to use the editor or IDE.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The reasons for implementation choices.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to run the software as an end user.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The location of test data.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether documentation is up to date.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether this is the latest version of the source code.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Workarounds for common problems.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to set up the development environment.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Please expand on your choice:

---

### 43. If the expert guesses answers to the newcomer’s questions, the newcomer will find this:

<table>
<thead>
<tr>
<th>Information</th>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to use the editor or IDE.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The reasons for implementation choices.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to run the software as an end user.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>The location of test data.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether documentation is up to date.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Whether this is the latest version of the source code.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Workarounds for common problems.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>How to set up the development environment.</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Please expand on your choice:

---

### Expert advice

**Expert:** an experienced developer who knows their codebase well.  
**Newcomer:** an experienced developer who does not know that codebase.  
**Session:** a time when the expert explains their code to the newcomer.
44. If the expert informs the newcomer about typically long-running tasks, the newcomer will find this:

- Very useful
- Useful
- Neither useful nor useless
- Useless
- Very useless

Please expand on your choice:

45. If the expert tells stories about issues encountered by the team before the newcomer joined, the newcomer will find this:

- Very useful
- Useful
- Neither useful nor useless
- Useless
- Very useless

Please expand on your choice:

46. If the expert informs the newcomer which areas of code are off-limits, the newcomer will find this:

- Very useful
- Useful
- Neither useful nor useless
- Useless
- Very useless

Please expand on your choice:

**After the session**

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

47. If the expert bases the session on the newcomer's development task, the newcomer will find this:

- Very useful
- Useful
- Neither useful nor useless
- Useless
- Very useless

Please expand on your choice:

48. If the newcomer explores the code independently after the session, this is:

- Very useful
- Useful
- Neither useful nor useless
- Useless
- Very useless

Please expand on your choice:

49. If the newcomer takes copies of diagrams used in the session, this is:

- Very useful
- Useful
- Neither useful nor useless
- Useless
- Very useless

Please expand on your choice:

**After the session**
50. If the newcomer takes notes in the session, this is:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

---

51. If the newcomer takes breaks during the session, this is:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

---

52. In the early stages, the newcomer will find that diagrams with a lot of detail are:

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

---

Final comments

53. Do you have any further comments about the issues raised in this questionnaire?

Thank you!

You have completed the questionnaire. Thank you very much for your participation in this research.

If you have any questions about this study, please contact rebecca.yates@lero.ie
Appendix E

Analysis of Fit Questionnaire

This section gives the results and analysis of the fit questionnaire for traceability. The conclusions of this analysis are described in Section 7.1.

E.1 Responses to fit questionnaire

In Table E.1, statements with clear support (those with a score of at least 11 and no more than 2 respondents in disagreement) have a white background. Those with significant disagreement (a score of 0 or less, or 3 or more disagreements) are coloured dark grey, and the remainder with ambiguous responses are coloured light grey.
<table>
<thead>
<tr>
<th>Question</th>
<th>Statement</th>
<th>Expected response</th>
<th>Support for expected response</th>
<th>Responses: expected</th>
<th>Responses: unexpected</th>
</tr>
</thead>
<tbody>
<tr>
<td>6a</td>
<td>It is useful to the newcomer if: The expert describes the high-level architecture.</td>
<td>Agreement</td>
<td>25</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>6b</td>
<td>It is useful to the newcomer if: The expert points out the design patterns in use in the code.</td>
<td>Agreement</td>
<td>17</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6c</td>
<td>It is useful to the newcomer if: The expert goes through all the code line-by-line.</td>
<td>Disagree</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>6d</td>
<td>It is useful to the newcomer if: The expert names chunks of code (e.g. &quot;the initialisation stuff&quot;).</td>
<td>Agreement</td>
<td>18</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>6e</td>
<td>It is useful to the newcomer if: The expert points out simple and complex areas of code.</td>
<td>Agreement</td>
<td>15</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>6f</td>
<td>It is useful to the newcomer if: The expert describes the clients.</td>
<td>Agreement</td>
<td>17</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>6g</td>
<td>It is useful to the newcomer if: The expert goes through complex code line-by-line.</td>
<td>Agreement</td>
<td>8</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>6h</td>
<td>It is useful to the newcomer if: The expert points out similarities or commonly used idioms in the code.</td>
<td>Agreement</td>
<td>20</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>6i</td>
<td>It is useful to the newcomer if: The expert points out costs (e.g. in terms of memory use, speed or file size).</td>
<td>Agreement</td>
<td>11</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Question</td>
<td>Statement</td>
<td>Expected response</td>
<td>Support for expected response</td>
<td>Responses: expected</td>
<td>Responses: unexpected</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>--------------------------------</td>
<td>--------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>8</td>
<td>If the newcomer drives, they are more engaged because they cannot 'switch off'.</td>
<td>Agreement</td>
<td>9</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>If the newcomer drives, the session is slower and less fluid.</td>
<td>Agreement</td>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>If the newcomer drives, this improves their ability to find code independently after the session.</td>
<td>Agreement</td>
<td>16</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>If the newcomer drives, it is hard for the expert to direct them (the expert may be 'back seat driving').</td>
<td>Agreement</td>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>If the newcomer drives, the expert can judge how well the newcomer is learning the system.</td>
<td>Agreement</td>
<td>13</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>If the newcomer drives, it is hard for the expert to navigate and locate code.</td>
<td>Agreement</td>
<td>1</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>If the newcomer drives, the expert is prevented from going too quickly through the system.</td>
<td>Agreement</td>
<td>11</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>If the newcomer drives, they will find the session more confusing.</td>
<td>Disagree</td>
<td>2</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>If the expert describes longer-term ongoing changes to the codebase, the newcomer will find this:</td>
<td>Useful</td>
<td>13</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>If the expert points out temporary fixes in the code, the newcomer will find this:</td>
<td>Useful</td>
<td>13</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Question</td>
<td>Statement</td>
<td>Expected response</td>
<td>Support for expected response</td>
<td>Responses: expected</td>
<td>Responses: unexpected</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------</td>
<td>--------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>18</td>
<td>If the expert points out dead code, the newcomer will find this:</td>
<td>Useful</td>
<td>8</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>Project conventions may change over time, resulting in two or more</td>
<td>Useful</td>
<td>17</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>conventions coexisting in the codebase. If the expert discusses this,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>the newcomer will find this:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>If the session uses the newcomer’s machine, it will be</td>
<td>Agreement</td>
<td>10</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>interrupted by missing configurations or data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>If the session uses the newcomer’s machine, the newcomer will not be</td>
<td>Agreement</td>
<td>19</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>blocked afterwards by environment problems.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>If the session uses the newcomer’s machine, the explanation is not as</td>
<td>Agreement</td>
<td>-3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>smooth.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Without a thorough understanding of (part of) the code, the expert</td>
<td>Agreement</td>
<td>-3</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>cannot present a high level view of it, and instead resorts to working</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>through code line by line.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>If the code is well commented, the expert will find this:</td>
<td>Useful</td>
<td>20</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>Experts have become so at home in the system that they forget to</td>
<td>Agreement</td>
<td>15</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>mention gotchas - quirks of the codebase or the environment with known</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>but non-obvious workarounds.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Question</td>
<td>Statement</td>
<td>Expected response</td>
<td>Support for expected response</td>
<td>Responses: expected</td>
<td>Responses: unexpected</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------</td>
<td>---------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>27</td>
<td>Experts find the debugger a useful tool for explanations.</td>
<td>Agreement</td>
<td>8</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>Experts remember how to set up the newcomer’s development environment.</td>
<td>Disagree</td>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>If the expert points out possible misconceptions (e.g. “it gets the information from the server, it doesn’t do it locally”), the newcomer will find this:</td>
<td>Useful</td>
<td>18</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>If the expert ends each chunk of explanation (e.g. ‘that’s all there is to it’) the newcomer will find this:</td>
<td>Useful</td>
<td>-11</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>31</td>
<td>If the expert uses the debugger to step through code, the newcomer will find this:</td>
<td>Useful</td>
<td>6</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>32</td>
<td>From the expert’s point of view, the ideal session provides the newcomer with as much information as possible.</td>
<td>Agreement</td>
<td>-5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>33</td>
<td>From the expert’s point of view, the ideal session allows the expert to return quickly to normal work tasks.</td>
<td>Agreement</td>
<td>-5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>34</td>
<td>From the expert’s point of view, the ideal session does not waste time on environment setup.</td>
<td>Agreement</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Question</td>
<td>Statement</td>
<td>Expected response</td>
<td>Support for expected response</td>
<td>Responses: expected</td>
<td>Responses: unexpected</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>------------------------------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>35</td>
<td>From the newcomer’s point of view, the ideal session engages the newcomer, even at the cost of frustration to the expert.</td>
<td>Agreement</td>
<td>19</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>From the newcomer’s point of view, the ideal session directly helps them to make early contributions.</td>
<td>Agreement</td>
<td>17</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>From the newcomer’s point of view, the ideal session will finish early if the newcomer has enough information to be going on with.</td>
<td>Agreement</td>
<td>13</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>38</td>
<td>From the newcomer’s point of view, the ideal session includes help with environment setup.</td>
<td>Agreement</td>
<td>16</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>From the newcomer’s point of view, the ideal session gets them past problems that would otherwise block them.</td>
<td>Agreement</td>
<td>24</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>40a</td>
<td>[The newcomer will always need to ask the expert about]: Whether documentation is up to date.</td>
<td>Agreement</td>
<td>8</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>40b</td>
<td>[The newcomer will always need to ask the expert about]: The reasons for implementation choices.</td>
<td>Agreement</td>
<td>15</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>40c</td>
<td>[The newcomer will always need to ask the expert about]: How to set up the development environment.</td>
<td>Agreement</td>
<td>-4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>40d</td>
<td>[The newcomer will always need to ask the expert about]: Workarounds for common problems.</td>
<td>Agreement</td>
<td>1</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Question</td>
<td>Statement</td>
<td>Expected response</td>
<td>Support for expected response</td>
<td>Responses: expected</td>
<td>Responses: unexpected</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------</td>
<td>---------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>40e</td>
<td>[The newcomer will always need to ask the expert about]: The location of test data.</td>
<td>Agreement</td>
<td>6</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>40f</td>
<td>[The newcomer will always need to ask the expert about]: How to run the software as an end user.</td>
<td>Agreement</td>
<td>-10</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>40g</td>
<td>[The newcomer will always need to ask the expert about]: Whether this is the latest version of the source code.</td>
<td>Agreement</td>
<td>-11</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>40h</td>
<td>[The newcomer will always need to ask the expert about]: How to use the editor or IDE.</td>
<td>Disagree</td>
<td>25</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>If the expert gives the limits of their own knowledge, the newcomer will find this:</td>
<td>Useful</td>
<td>7</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>If the expert gives their opinion on the qualities of information sources, the newcomer will find this:</td>
<td>Useful</td>
<td>14</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>43</td>
<td>If the expert guesses answers to the newcomer’s questions, the newcomer will find this:</td>
<td>Useless</td>
<td>10</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>44</td>
<td>If the expert informs the newcomer about typically long-running tasks, the newcomer will find this:</td>
<td>Useful</td>
<td>14</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>If the expert tells stories about issues encountered by the team before the newcomer joined, the newcomer will find this:</td>
<td>Useful</td>
<td>12</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Question</td>
<td>Statement</td>
<td>Expected response</td>
<td>Support for expected response</td>
<td>Responses: expected</td>
<td>Responses: unexpected</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-------------------------------</td>
<td>---------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>46</td>
<td>If the expert informs the newcomer which areas of code are off-limits, the newcomer will find this:</td>
<td>Useful</td>
<td>9</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>47</td>
<td>If the expert bases the session on the newcomer’s development task, the newcomer will find the session:</td>
<td>Useful</td>
<td>21</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>If the newcomer explores the code independently after the session, this is:</td>
<td>Useful</td>
<td>25</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>If the newcomer takes copies of diagrams used in the session, this is:</td>
<td>Useful</td>
<td>22</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>If the newcomer takes notes in the session, this is:</td>
<td>Useful</td>
<td>21</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>If the newcomer takes breaks during the session, this is:</td>
<td>Useful</td>
<td>9</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>In the early stages, the newcomer will find that diagrams with a lot of detail are:</td>
<td>Useless</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
E.2 Unclear and unsupported statements

Statement 6c: “It is useful to the newcomer if: The expert goes through all the code line-by-line.”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.2: Support for statement 6c

As expected, most respondents disagreed with this statement on the basis that it was unnecessary for experienced developers and/or would overload them with unnecessary details. Instead, they recommend a top-down approach.

No changes are required.

Statement 6g: “It is useful to the newcomer if: The expert goes through complex code line-by-line.”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table E.3: Support for statement 6g

In the comments, respondents agree that this can occasionally be useful, but that too much detailed explanation tends to overload the newcomer. (In addition, code that requires this level of explanation is likely to be poorly written.) A high-level explanation is considered a prerequisite.

Going through any code line-by-line with a newcomer can be useful but should be considered a last resort.

Statement 8: “If the newcomer drives, they are more engaged because they cannot ‘switch off’.”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table E.4: Support for statement 8
APPENDIX E. ANALYSIS OF FIT QUESTIONNAIRE

The experience gained from navigating the code, and the need to ask questions to make progress, are seen as advantages for the newcomer’s learning process. Respondents also note that this is difficult for the newcomer and may make them self-conscious.

At least one respondent conflated ‘driving’ with ‘directing’ the session and argued that the newcomer would lack the knowledge to direct the session. This supports the claim that newcomers need prior knowledge of the codebase before they can effectively self-direct their investigations, but is not relevant to the statement under discussion.

No changes are required.

Statement 9: “If the newcomer drives, the session is slower and less fluid”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table E.5: Support for statement 9

All comments agree that this apparently negative effect is normal and desirable when explaining to a newcomer.

No changes are required.

Statement 11: “If the newcomer drives, it is hard for the expert to direct them (the expert may be ‘back seat driving’).”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.6: Support for statement 11

Two main points emerge from the comments. The first is that experts do find “back seat driving” extremely frustrating. However, given the newcomer’s experience as a developer, respondents also question the need for back-seat driving. Once the newcomer has grasped the general overview and terminology of the system, the expert can provide high-level
direction to code and allow the newcomer to locate the code in their own way.

Back-seat driving is frustrating for the expert and this concern may prevent experts from allowing newcomers to drive. However, if the newcomer cannot find the code without this low-level direction, it may indicate that they do not yet have a high-level grasp of the system or terminology.

**Statement 13: “If the newcomer drives, it is hard for the expert to navigate and locate code.”**

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.7: Support for statement 13

The comments indicate a strong link between this issue and ‘back seat driving’ (statement 11). This problem is exacerbated if the expert is underprepared.

Experts are more likely to resort to back-seat driving if they are themselves unfamiliar with the code and need to explore it on-the-fly.

**Statement 15: “If the newcomer drives, they will find the session more confusing.”**

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.8: Support for statement 15

The comments for this statement do not show any consensus. This is interpreted to mean that newcomer confusion has many possible causes that are unrelated to the driver role.

No changes are required.

**Statement 18: “If the expert points out dead code, the newcomer will find this:”**
Appendix E. Analysis of FIT Questionnaire

Responses fall into two groups, with one stating that there is little use in wasting time on comprehending dead code (and that it should be removed), and the other that this can prompt useful discussions on why the code was removed.

Pointing out dead code is useful only if the expert is giving the reasons for the change and not simply stating that the code is unused.

Statement 21: “If the session uses the newcomer’s machine, it will be interrupted by missing configurations or data.”

The disagreeing respondent comments “All code and project data should come from source control system” (S3 Newcomer) and several other comments agree that the setup should be documented or standardised. However, most comments concede that this idealised situation rarely occurs in practice and, that being the case, such interruptions are helpful to the newcomer because they uncover blocking problems while the expert is on-hand to assist.

No changes are required.

Statement 23: “If the session uses the newcomer’s machine, the explanation is not as smooth.”

Table E.9: Support for statement 18

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.10: Support for statement 21

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.11: Support for statement 23

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>
Similarly to Statement 34, respondents protest that while this may be the case, it is more important to fix the newcomer’s development environment and answer their questions than to have a ‘smooth’ session. This implies that this concern is not the reason for the rarity of use of the newcomer’s machine in the sessions.

While experts may be frustrated by session problems, a ‘smooth’ session is not as useful to the newcomer as one that runs into and then addresses their problems and misunderstandings. Environment setup in particular may feel like a ‘problem’ in the context of an explanation session, but is an important prerequisite for effective explanation.

Statement 24: “Without a thorough understanding of (part of) the code, the expert cannot present a high level view of it, and instead resorts to working through code line by line.”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Table E.12: Support for statement 24

Most comments object to the idea that the expert must necessarily understand all the code, and state that an expert can provide a high-level view without knowing all the low-level details. On reflection, it is clear that this is the case, so the line-by-line just-in-time-comprehension behaviour occasionally seen must have a different cause.

It is not the lack of thorough low-level detail that causes this behaviour in experts, but the lack of a high-level overview of the system. Similarly, by passing on a high-level view of the system to the newcomers, the experts also reduce the newcomers’ need to examine the details of the code.

Statement 27: “Experts find the debugger a useful tool for explanations.”

Respondents suggested that debugger features could be useful for tracing flow through highly delocalised code, for complex code, or for the expert to refresh and organise their own understanding. However, most comments did not offer strong support for debugger features in sessions being useful.
APPENDIX E. ANALYSIS OF FIT QUESTIONNAIRE

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.13: Support for statement 27

to newcomers.

This statement was expected to generate agreement, but in light of the strength of the support for high-level over low-level explanation, the usefulness of debugger features in explanations is unclear. No changes.

**Statement 28:** “Experts remember how to set up the newcomer’s development environment.”

Table E.14: Support for statement 28

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

The comments indicate that, although experts ‘should’ remember or the information ‘should’ be documented, this is rarely the case. Instead, experts have often forgotten the details, holding up the newcomer’s progress, although they may remember useful information when presented with the errors. This supports the claim that environment setup is a problem for both newcomers and experts.

No changes are required.

**Statement 30:** “If the expert ends each chunk of explanation (e.g. ”that’s all there is to it”) the newcomer will find this:”

Table E.15: Support for statement 30

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

This question received strong disagreement but with only two comments, one focused on the overuse of phrases, and the other sceptical that any part of a system can be fully explained in isolation.
This statement was intended to check whether this explanation strategy allowed newcomers to assign a measure of size to the code implementing features. It is not clear that the wording made this effect clear. No changes are made.

**Statement 31: “If the expert uses the debugger to step through code, the newcomer will find this:”**

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.16: Support for statement 31

Respondents thought newcomers might find this useful for complex or delocalised code, or to learn the filenames of relevant files, but overall the comments were negative. In general, this technique is considered too detailed for explanation sessions.

Stepping through code using the debugger is only useful to the newcomer for complex or delocalised code.

**Statement 32: “From the expert’s point of view, the ideal session provides the newcomer with as much information as possible.”**

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.17: Support for statement 32

Most of the comments do not address the question as asked, but instead argue that this strategy risks overloading the newcomer and instead the session should provide a minimum of information to get the newcomer started on a task. Too much information is inefficient for the expert as well, because it will not be retained by the newcomer and so the expert may have to explain some areas more than once.

No changes are required.
APPENDIX E. ANALYSIS OF FIT QUESTIONNAIRE

Statement 33: “From the expert’s point of view, the ideal session allows the expert to return quickly to normal work tasks.”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.18: Support for statement 33

Respondents disagreed, arguing that onboarding newcomers is also a ‘normal work task’ and that although onboarding efficiency is desirable, rushing through a session is unlikely to be efficient in the long run.

This reinforces the changes outlined in Statement 34: some of the expert’s choices about the session are not done to save time for the expert’s short term benefit, but for some other reason.

Statement 34: “From the expert’s point of view, the ideal session does not waste time on environment setup.”

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.19: Support for statement 34

There is some support for this ‘ideal’ case, in which the newcomer is able to follow up-to-date instructions to set up their environment without help. However, many comments recognise that in practice, this support is not available. In this case, time that the expert spends on the problem is not considered a waste since it is necessary for the newcomer to become valuable (and, more tellingly, to prevent repeated interruptions from the newcomer). It is also an opportunity for developers to improve the setup documentation for subsequent newcomers.

The comments do not match the realities of the observed sessions, suggesting that although experts recognise the newcomer’s difficulties with setup, they are reluctant to help. One reason for this, implied by Statement 28, is that experts often do not remember the details of the setup.
The expert’s reluctance to help with environment setup is not because they consider it a waste of their time. Instead, either the expert believes (usually mistakenly) that the setup instructions are adequate, or they have difficulty providing assistance because they have forgotten the details.

**Statement 40a:** “[The newcomer will always need to ask the expert about]: Whether documentation is up to date.”

<table>
<thead>
<tr>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

Table E.20: Support for statement 40a

As expected, most respondents agreed that expert help would be needed to establish the reliability of the documentation. Some reported that this could easily be done without help but did not offer their strategies in the comments, although one comment, “Documentation is never up to date” implies a strategy of simply never trusting any documentation.

The reasons for the unexpected responses are unclear, so no changes are possible. There may be more to discover here.

**Statement 40c:** “[The newcomer will always need to ask the expert about]: How to set up the development environment.”

<table>
<thead>
<tr>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table E.21: Support for statement 40c

As with Statement 40d, respondents point out that this depends on the availability and quality of project documentation.

The newcomer will typically need expert help to set up their environment unless this has been documented and kept updated.

**Statement 40d:** “[The newcomer will always need to ask the expert about]: Workarounds for common problems.”
APPENDIX E. ANALYSIS OF FIT QUESTIONNAIRE

<table>
<thead>
<tr>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.22: Support for statement 40d

As with most statements in this group, this is considered to depend heavily on the availability and quality of project documentation. Some respondents interpreted this question as ‘universal common problems’ of the sort that could be solved by searching online.

The newcomer will need to ask the expert for project-specific workarounds unless the expert points them to project documentation of workarounds.

Statement 40e: “[The newcomer will always need to ask the expert about]: The location of test data.”

<table>
<thead>
<tr>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Table E.23: Support for statement 40e

This statement is supported, but a common theme in the comments is that this information should be included in the documentation.

No changes are required.

Statement 40f: “[The newcomer will always need to ask the expert about]: How to run the software as an end user.”

<table>
<thead>
<tr>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.24: Support for statement 40f

As with other questions in this group, this depends on project documentation. Respondents point out that end-user documentation
should always be available. However, in the observed sessions, end-user documentation was either unavailable (because the system was new or experimental) or the newcomers were not shown where to find it.

It is useful for newcomers if experts either point out end-user documentation or explain how to run the software.

**Statement 40g:** “[The newcomer will always need to ask the expert about]: Whether this is the latest version of the source code.”

<table>
<thead>
<tr>
<th>Easy to find out without expert help</th>
<th>Possible to find out without expert help</th>
<th>Usually need to ask an expert</th>
<th>Always need to ask an expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.25: Support for statement 40g

Respondents disagree that this is a difficulty for experienced developers, provided they have access to the version control system. However, in the sessions, code version issues were occasionally caused by very recent or simultaneous maintenance activities.

As experienced developers, newcomers are able to compare their local codebase with that under version control. However, while version control systems make the problem manageable, they do not completely eradicate temporal issues caused by concurrent maintenance or poor version control discipline.

**Statement 41:** “If the expert gives the limits of their own knowledge, the newcomer will find this:”

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.26: Support for statement 41

Some responses focus on the fact that this information does not help the newcomer to understand the code, and is therefore not useful. However, the majority of comments recognise that this contributes to the newcomer’s information seeking strategy and is therefore somewhat useful.
APPENDIX E. ANALYSIS OF FIT QUESTIONNAIRE

No changes are required.

Statement 43: “If the expert guesses answers to the newcomer’s questions, the newcomer will find this:”

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Table E.27: Support for statement 43

Respondents were expected to report that guessing answers would confuse the newcomer. However, two respondents point out that this information might still be useful as long as it is clear that it is the expert’s guess. Most comments agree that finding out the answer or finding a more knowledgeable co-worker is preferable to guessing.

These recommendations are added to the theory.

Statement 46: “If the expert informs the newcomer which areas of code are off-limits, the newcomer will find this:”

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table E.28: Support for statement 46

Many respondents questioned the idea of ‘off-limits’ code and argued that this limitation would reduce understanding, innovation and motivation. However, code ownership practices depend on company culture, and changes would be outside the scope of an onboarding session (and of this study). Other respondents argue that information about limitations helps to keep the newcomer focused on their task and avoid wasting time on unnecessary work.

Information about limitations is useful to the newcomer, but should be accompanied by reasons for the limitations (for example, whether a limitation is due to permanent code ownership conventions or current priorities).
### Table E.29: Support for statement 51

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Statement 51: “If the newcomer takes breaks during the session, this is:”

This statement is supported, but many comments argue that a session long enough to require breaks is probably too long. This implicitly supports the idea that sessions should end when the newcomer has enough information to get started, rather than trying to transmit as much as possible.

No changes are required.

### Table E.30: Support for statement 52

<table>
<thead>
<tr>
<th>Very useful</th>
<th>Useful</th>
<th>Neither useful nor useless</th>
<th>Useless</th>
<th>Very useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Statement 52: “In the early stages, the newcomer will find that diagrams with a lot of detail are:”

Support for this statement is mixed, but the comments provide further support for a top-down approach to explanations, beginning with high-level ‘overview’ diagrams and only providing details as required for the newcomer’s task.

No changes are required.

### E.3 Other notable statements

As expected, respondents disagreed strongly with Statement 40h (“[The newcomer will always need to ask the expert about]: How to use the editor or IDE.”). This demonstrates that respondents were considering newcomers (who, as experienced developers would not need this help from the expert) and not novice developers who might require this kind of tuition.
Respondents were also asked to recommend strategies for assigning the role of driver and owner. Their responses indicate that the expert should take the driver role initially, but hand over to the newcomer once they have grasped the high level view of the system (Table E.31) and that the session should use the newcomer’s machine (Table E.32). These recommendations are discussed further in Section 9.3.

Table E.31: Support for driver recommendation

<table>
<thead>
<tr>
<th>In the session, who should drive (i.e. have control of the keyboard and mouse)?</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>The expert should drive initially and provide a high level overview, and then hand over to the newcomer.</td>
<td>11</td>
</tr>
<tr>
<td>The expert should drive.</td>
<td>3</td>
</tr>
<tr>
<td>It depends on personal preference of both expert and newcomer.</td>
<td>1</td>
</tr>
<tr>
<td>The newcomer should drive (guided by the expert).</td>
<td>0</td>
</tr>
<tr>
<td>It doesn’t matter.</td>
<td>0</td>
</tr>
<tr>
<td>Other (please specify)</td>
<td>0</td>
</tr>
</tbody>
</table>

Table E.32: Support for owner recommendation

<table>
<thead>
<tr>
<th>Whose machine should be used in the session?</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>The newcomer’s machine.</td>
<td>9</td>
</tr>
<tr>
<td>It doesn’t matter.</td>
<td>3</td>
</tr>
<tr>
<td>The expert’s machine.</td>
<td>1</td>
</tr>
<tr>
<td>Both the expert’s and newcomer’s machines (assuming they can be placed side by side)</td>
<td>1</td>
</tr>
<tr>
<td>Other (“probably the newcomer’s”)</td>
<td>1</td>
</tr>
<tr>
<td>A third machine.</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix F

Testing Generalisation
Questionnaire

The following pages comprise the questionnaire that was used to evaluate the generalisation of the theory. The questions were presented online on SurveyMonkey.com and completed in the participants’ own time. The development of this questionnaire, including question logic not shown here, is discussed in Section 7.2. Note that to save space, page breaks are not represented.
Explaining source code

Introduction

This questionnaire is part of a study on onboarding in software engineering. Overall, this research aims to understand how newcomers are introduced to the codebase, and to suggest improvements to the process.

This online questionnaire should take no more than 25 minutes to complete. Your full questionnaire answers will not be shared with anyone outside the research group. Your responses will be used in the evaluation of this research, and text responses may be anonymously quoted.

You and the data you provide will be anonymized in all reports using this data.

You have the right not to answer questions and to withdraw at any time. By continuing with this questionnaire you are deemed to have given consent to participate. If you have any questions, please contact the researcher. If you have any concerns, please contact the Chair of the Science & Engineering Research Ethics Committee at the University of Limerick.

Contact information:
Researcher: Rebecca Yates (rebecca.yates@lero.ie)
Supervisors: Jim Buckley (jim.buckley@ul.ie) and Norah Power (norah.power@ul.ie)
Chair of S&E REC: Dr Thomas Waldman (+353 61 202802)

About you

1. This questionnaire is part of a larger study on code explanation. We are looking for responses from developers who haven't already been a part of this study.

Have you already participated in this study? (observations, interviews, questionnaires, etc)

- Yes
- No

Thank you!

Thank you for your help with this study! This questionnaire is aimed at developers who have not previously been involved in the research. If you know of any developers who might be willing to take the questionnaire, please send them the link (https://www.surveymonkey.com/s/MLJF78K).

About your organisation

Please answer for your current or most recent project/organisation.
2. Is your code open or closed source?
- Open source
- Closed source
- A mixture
Other (please specify)

3. Where does your team work?
- In the same building as me
- In different geographic locations
- Some in the same building, some in other locations
Other (please specify)

4. What type of project do you work on?
- Commercial
- Academic
- Civil/governmental
Other (please specify)

5. How many people work at your organisation? (approximately):

6. Which country do you work in?

About you

7. How much commercial software development experience do you have? (in years)

8. How much other software development experience (e.g. personal projects) do you have? (in years)

9. How many different software projects have you worked on?

Getting newcomers up to speed
10. In your current/most recent project, when a newcomer joins the team, does an expert give them an in-person walkthrough of the code?
- Always
- Usually
- Sometimes
- Rarely
- Never

Please expand on your choice:

Getting newcomers up to speed

11. In your current/most recent project, how do newcomers usually familiarise themselves with the code?

12. Do you think that in-person walkthroughs of the code would be helpful? Why or why not?

Your experiences

*13. How recently have you explained a codebase to another developer?
- I have never done this
- I did this (days, months or years) ago:

Your experiences

*14. How recently has another developer explained a codebase to you?
- I have never had this happen
- This happened (days, months or years) ago:
15. When you need to understand unfamiliar code, what strategies do you use?

16. When another developer needs to understand your code, what have you done to help them?

17. In either situation, do you think that in-person explanation of code would be helpful?

Your experiences

*18. How recently has another developer explained a codebase to you?

- I have never had this happen
- This happened (days, months or years) ago: [ ]

Instructions
In this questionnaire, “expert” refers to an experienced developer who knows their codebase well, and “newcomer” refers to an experienced developer (not a novice) who does not know that codebase. The questions refer to sessions in which an expert explains the code to a newcomer in person, so that the newcomer can begin development work.

Each multiple-choice question in this questionnaire is followed by a comment box. Comments are optional, but please expand on your choice if you feel strongly about the issue, or if the available choices don't adequately cover your response.

Session configuration

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

19. **Usually, sessions are conducted on:**
   - The expert's machine.
   - The newcomer's machine.
   - A third machine.
   - The expert and the newcomer use their own machines, side by side.
   - Other (please describe)

   Please expand on your choice:

20. **Usually:**
   - The expert guides the newcomer around the system.
   - The newcomer looks around the system and the expert provides commentary.
   - Other (please describe)

   Please expand on your choice:

21. **Usually, the mouse and keyboard are operated by:**
   - The expert
   - The newcomer
   - The expert to start with, then the newcomer
   - They each operate their own machine
   - Other (please describe)

   Please expand on your choice:
22. In your opinion, does the typical setup work well, or could it be improved?

Session timing

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.

23. Sessions usually take place:

- Before the newcomer has been given a development task.
- After the newcomer has been given a development task.
- The newcomer is given a development task during the session.
- Whenever the expert is available (irrespective of the newcomer's task).
- Other (please describe)

Please expand on your choice:

24. Usually, sessions take place:

- Before the newcomer's development environment is set up.
- While the newcomer is still setting up their environment.
- After the newcomer's development environment is set up.
- Part of the session is to help the newcomer set up their environment.
- Whenever the expert is available (irrespective of the newcomer's environment)
- Other (please describe)

Please expand on your choice:

25. In your opinion, are sessions typically well-timed, or could this be improved?

Typical session content

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
26. Sessions include a high-level look at the components making up the system.

27. Sessions include tracing the flow of execution through selected parts of the code.

28. Sessions include discussion of the requirements and design principles that led to the current design.

29. Sessions include discussion of how previous, current and future development work is impacting the code.

30. Sessions include demonstrating the software from an end-user point of view.

31. What else takes place in a session?

The expert's point of view

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
### 32. Before explaining code to a newcomer, do you do any preparation? (If so, what do you do?)

### 33. It’s common for the expert to forget how to set up the development environment.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

### 34. It’s common for the expert to be unfamiliar with some of the code.

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please expand on your choice:

### 35. Ideally, what should experts do differently to improve the sessions?


### The newcomer’s point of view

- **Expert**: an experienced developer who knows their codebase well.
- **Newcomer**: an experienced developer who does not know that codebase.
- **Session**: a time when the expert explains their code to the newcomer.
36. During the session, it's common for the newcomer to have problems with: (check all that apply)

- The project terminology
- Navigating the project
- Too much information
- Not enough information
- Information that is not relevant to their task
- Unclear names in code
- The expert going too fast through the system
- The expert going too slowly through the system
- Other (please describe)

Please expand on your choice:

37. After the session, it’s common for the newcomer to have problems with: (check all that apply)

- Remembering the content of the session
- Locating code
- Locating documentation
- Development environment setup
- Unclear names
- Missing test data
- Being unsure what to do
- Other work taking precedence over work on the new system
- Problems running the software as an end user
- Difficulty using the IDE
- Difficulty using the language

Please expand on your choice:

Useful activities

Expert: an experienced developer who knows their codebase well.
Newcomer: an experienced developer who does not know that codebase.
Session: a time when the expert explains their code to the newcomer.
38. For the newcomer, the most useful activities in a session are: (check all that apply)

- Getting advice about the newcomer's next development task
- General advice about how to work effectively on this system
- Fixing problems with the newcomer's development environment
- Going through complex code line-by-line
-Getting a high-level overview of the components of the system
- Learning the project terminology
- Finding out where documentation is to be found
- Learning whether the documentation is up to date
- Learning who to ask about different areas of the system
- Going through all the code line-by-line
- Finding out reasons for the implementation choices
- Learning to use the IDE
- Learning the programming language
- Finding out how the code is likely to change in the near future
- Learning about project naming conventions

Please expand on your choices:

39. How could a typical session be improved for the newcomer?

Final comments

40. Do you have any further comments about the issues raised in this questionnaire?

Thank you!
You have completed the questionnaire. Thank you very much for your participation in this research.

If you have any questions about this study, please contact rebecca.yates@lero.ie
Appendix G

Analysis of Generalisation Questionnaire

The data in this section result from the questionnaire shown in Appendix F. The analysis is discussed in Section 7.2.

G.1 Respondent background

This section details the respondents’ background as software developers.

<table>
<thead>
<tr>
<th>Codebase types</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closed source</td>
<td>17</td>
</tr>
<tr>
<td>Open source</td>
<td>7</td>
</tr>
<tr>
<td>A mixture</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Team locations</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the same building as me</td>
<td>16</td>
</tr>
<tr>
<td>Some in the same building, some in other locations</td>
<td>9</td>
</tr>
<tr>
<td>In different geographic locations</td>
<td>6</td>
</tr>
</tbody>
</table>
### Table G.3: Project types

<table>
<thead>
<tr>
<th>What type of project do you work on?</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial</td>
<td>26</td>
</tr>
<tr>
<td>Academic</td>
<td>4</td>
</tr>
<tr>
<td>Civil/governmental</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table G.4: Organisation sizes

<table>
<thead>
<tr>
<th>How many people work at your organisation? (answers grouped)</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 or fewer</td>
<td>8</td>
</tr>
<tr>
<td>26 - 100</td>
<td>7</td>
</tr>
<tr>
<td>101 - 250</td>
<td>9</td>
</tr>
<tr>
<td>250 - 1000</td>
<td>3</td>
</tr>
<tr>
<td>More than 1000</td>
<td>4</td>
</tr>
</tbody>
</table>

Smallest organisation: 6 people
Largest organisation: 30,000 people

### Table G.5: Countries

<table>
<thead>
<tr>
<th>Which country do you work in?</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK</td>
<td>16</td>
</tr>
<tr>
<td>Ireland</td>
<td>5</td>
</tr>
<tr>
<td>USA</td>
<td>4</td>
</tr>
<tr>
<td>Belgium</td>
<td>1</td>
</tr>
<tr>
<td>Brazil</td>
<td>1</td>
</tr>
<tr>
<td>India</td>
<td>1</td>
</tr>
<tr>
<td>Italy</td>
<td>1</td>
</tr>
<tr>
<td>Pakistan</td>
<td>1</td>
</tr>
<tr>
<td>USA/Canada</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table G.6: Experience

<table>
<thead>
<tr>
<th>How much software development experience do you have? (in years)</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial</td>
<td>0.5</td>
<td>30</td>
<td>9.4</td>
</tr>
<tr>
<td>Other, e.g. personal projects</td>
<td>1.5</td>
<td>35</td>
<td>11.7</td>
</tr>
</tbody>
</table>

### G.2 Support for Generalisation of Theory

This section details the analysis of responses from the 17 respondents who had experience of onboarding sessions at their current or most recent
APPENDIX G. ANALYSIS OF GENERALISATION QUESTIONNAIRE

Table G.7: Number of projects

<table>
<thead>
<tr>
<th>How many different software projects have you worked on? (answers grouped)</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-5</td>
<td>4</td>
</tr>
<tr>
<td>6-10</td>
<td>7</td>
</tr>
<tr>
<td>11-20</td>
<td>7</td>
</tr>
<tr>
<td>21-50</td>
<td>3</td>
</tr>
<tr>
<td>more than 50</td>
<td>5</td>
</tr>
<tr>
<td>too many, lost count etc.</td>
<td>5</td>
</tr>
</tbody>
</table>

Table G.8: Prevalence of onboarding sessions

<table>
<thead>
<tr>
<th>In your current/most recent project, when a newcomer joins the team, does an expert give them an in-person walkthrough of the code?</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>6</td>
</tr>
<tr>
<td>Usually</td>
<td>4</td>
</tr>
<tr>
<td>Sometimes</td>
<td>7</td>
</tr>
<tr>
<td>Rarely</td>
<td>10</td>
</tr>
<tr>
<td>Never</td>
<td>4</td>
</tr>
</tbody>
</table>

Table G.9: Experience of ‘expert’ role (within ‘onboarding experience’ group)

<table>
<thead>
<tr>
<th>How recently have you explained a codebase to another developer? (answers grouped)</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within 1 day</td>
<td>3</td>
</tr>
<tr>
<td>Within 1 week</td>
<td>3</td>
</tr>
<tr>
<td>Within 1 month</td>
<td>4</td>
</tr>
<tr>
<td>More than 1 month ago</td>
<td>4</td>
</tr>
<tr>
<td>Unclear answer</td>
<td>3</td>
</tr>
</tbody>
</table>

Table G.10: Experience of ‘newcomer’ role (within ‘onboarding experience’ group)

<table>
<thead>
<tr>
<th>How recently has another developer explained a codebase to you? (answers grouped)</th>
<th>Respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within 1 day</td>
<td>2</td>
</tr>
<tr>
<td>Within 1 week</td>
<td>5</td>
</tr>
<tr>
<td>Within 1 month</td>
<td>3</td>
</tr>
<tr>
<td>More than 1 month ago</td>
<td>6</td>
</tr>
<tr>
<td>Unclear answer</td>
<td>1</td>
</tr>
</tbody>
</table>
workplace.

G.2.1 How experts represent code to newcomers

The theory describes the experts presenting four views of the code: Structural, Algorithmic, Rationale and Temporal. Accordingly, respondents were asked whether they recognised these four activities in their sessions and whether any other activities took place. As before, their responses are analysed for support of the theory and their comments are considered.

**Structural view**

The Structural view is well-supported (Table G.11). Among the neutral comments, one reports that this does not always happen, and another that it “depends on the newcomer’s role, team and competency”.

Table G.11: Support for Structural view

| Sessions include a high-level look at the components making up the system. |
|-----------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Strongly agree              | Agree           | Neither agree nor disagree | Disagree        | Strongly disagree | No response    |
| 8                           | 4               | 4                | 0               | 0               | 1               |

**Algorithmic view**

The Algorithmic view is generally supported (Table G.12). One respondent disagrees, reporting that detailed code review is left for the newcomer to carry out.

Table G.12: Support for Algorithmic view

| Sessions include tracing the flow of execution through selected parts of the code. |
|-----------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Strongly agree              | Agree           | Neither agree nor disagree | Disagree        | Strongly disagree |
| 4                           | 9               | 3                | 1               | 0               |
Rationale view

The Rationale view is mostly supported (Table G.13). One supporter added that this is more common when the design is unusual or new. Another describes the newcomer’s ‘why’ questions that drive this aspect of the explanation:

Respondent, in questionnaire: [Agree, because:] This tends to come out in the newcomer’s questions. “Why are module X and Y separate?” is a typical question, often answered by “because X is re-used elsewhere”, or “because the interface for Y can be implemented by Z in other cases”.

Table G.13: Support for Rationale view

<table>
<thead>
<tr>
<th>Sessions include discussion of the requirements and design principles that led to the current design.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly agree</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

The same respondent also describes the possible tensions in presenting the Rationale view:

Respondent, in questionnaire: There needs to be sensitivity between the [expert and newcomer] over the state of the code: the newcomer should not say accusingly “this looks like a hack”, yet the expert must be willing to admit “this is a bit of a hack”, and respond to the newcomer’s sensible questions with grace: “why is it done this way?” The answer is often along the lines of: “for historical reasons but we haven’t had chance to go back and clear it up”. I’m convinced every codebase has corners like this.

However, one respondent disagrees (with no comments) and five neither agree nor disagree. Of these five, one reports that it depends on the newcomer’s previous experience, and another that this may not occur because the experts lack this knowledge:

Respondent, in questionnaire: [Neither agree nor disagree, because:] Parts of the codebase are so old that we don’t really know ourselves.
APPENDIX G. ANALYSIS OF GENERALISATION QUESTIONNAIRE

The latter quote supports a later part of the theory, where experts’ lack of knowledge is one of the issues with onboarding sessions. Thus, the theory is amended to show the impact of this problem on delivery of the Rationale view.

Temporal view

There is some support for the value of Temporal information to the newcomer (Table G.15), but support for its occurrence in sessions is varied (Table G.14).

Table G.14: Support for Temporal view

<table>
<thead>
<tr>
<th>Sessions include discussion of how previous, current and future development work is impacting the code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly agree</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Table G.15: Support for value of Temporal information

<table>
<thead>
<tr>
<th>For the newcomer, the most useful activities in a session are:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding out how the code is likely to change in the near future</td>
<td>7</td>
</tr>
<tr>
<td>Learning whether the documentation is up to date</td>
<td>6</td>
</tr>
</tbody>
</table>

Some respondents consider this an important part of the session, for example:

Respondent, *in questionnaire*: [Strongly agree, because:] It is important to specify the direction we are taking and for what reason so it can be followed.

Others report that a Temporal view may not be necessary, depending on code ownership policy, team size or rate of change:

Respondent, *in questionnaire*: [Neither agree nor disagree, because:] As a small team (2-3 people) the impact of development doesn’t reach a speed which exceeds the capacity to understand the code as a whole.

280
Respondent, in questionnaire: [Disagree, because:] We are a small team, so we tend to work on very disjoint areas of the source code at any given time.

This is surprising, because the initial observations showed temporal information being discussed in a variety of company sizes and across varied code ownership practices. This demonstrates that there may be more factors influencing the presentation of the Temporal view. Further investigation of these possibilities is left for future work.

Other views

Respondents were also asked “What else takes place in a session?” This question was intended to check that no major activities were missed. Respondents mention task-specific advice, introduction to coworkers, documentation, project and company conventions, general development advice, architectural overview, discussion of requirements and newcomer questions, all of which are described in the theory. The only surprising activity mentioned was “opportunistic bug fixing” which was not seen in any of the sessions. This indicates that the theory provides good coverage of session activity.

G.2.2 Supporting newcomers in locating information

This aspect of the theory describes actions that experts take in order to shape the newcomer’s information seeking strategy to match project-specific sources. Experts point out documentation, co-workers, reference materials, code and code artifacts, and also give their opinions about their quality. Project terminology is a prerequisite for making use of these sources.

The responses support this theory (Table G.16), with particular importance given to co-workers as an information source; this matches the reports from the original participants. Project terminology is also considered important, while documentation has some support.

Respondent, in questionnaire: [Activities in a session include:] Introduction to other team members, documentation, policy and procedure, coding standards and practices...
Table G.16: Support for shaping information strategy

<table>
<thead>
<tr>
<th>Activity</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning who to ask about different areas of the system</td>
<td>13</td>
</tr>
<tr>
<td>Learning the project terminology</td>
<td>12</td>
</tr>
<tr>
<td>Finding out where documentation is to be found</td>
<td>10</td>
</tr>
<tr>
<td>Learning about project naming conventions</td>
<td>7</td>
</tr>
<tr>
<td>Learning whether the documentation is up to date</td>
<td>6</td>
</tr>
</tbody>
</table>

G.2.3 Supporting newcomers in making contributions

This section of the theory covers the advice that experts give to newcomers. This advice falls into two categories: general project-specific advice, and advice that is related to the newcomer’s current task (if they have one). There is support for the value of these activities (Table G.17).

Comments from respondents indicate that experts recognise the value of relevant, task-based information, for example:

Respondent, *in questionnaire*: They are mentored/coached on aspects of the module they are working on, rather than the whole codebase.

Table G.17: Support for advice-giving activity

<table>
<thead>
<tr>
<th>Activity</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting advice about the newcomer’s next development task</td>
<td>9</td>
</tr>
<tr>
<td>General advice about how to work effectively on this system</td>
<td>9</td>
</tr>
</tbody>
</table>

G.2.4 Problems with onboarding sessions

The theory identifies several common problems with onboarding sessions, which are attributed to session configuration, timing and content, expert preparation and the newcomer’s development environment.
Session configuration

The session configuration theory describes how sessions can be characterised in terms of three roles: the owner (who owns the machine in use), the director (who guides the content of the session) and the driver (who operates the keyboard and mouse). In most of the observed sessions, the expert takes all three roles, but the theory links this practice to the newcomer having continued environment setup difficulties and suffering from information overload.

Amongst these respondents, it is more common to use the newcomer’s machine (Table G.18). This is a significant difference, and the theory is amended to remove the statement that either owner configuration is more common.

Table G.18: Support for session configuration (owner role)

<table>
<thead>
<tr>
<th>Usually, sessions are conducted on:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>The newcomer’s machine</td>
<td>9</td>
</tr>
<tr>
<td>The expert and the newcomer use their own machines, side by side.</td>
<td>2</td>
</tr>
<tr>
<td>The expert’s machine</td>
<td>1</td>
</tr>
<tr>
<td>A third machine</td>
<td>1</td>
</tr>
<tr>
<td>“Expert machine via screensharing software”</td>
<td>1</td>
</tr>
<tr>
<td>“50/50 split”</td>
<td>1</td>
</tr>
<tr>
<td>“Not important”</td>
<td>1</td>
</tr>
<tr>
<td>“Initial run through on experts machine, more deep dive on newcomer’s machine”</td>
<td>1</td>
</tr>
</tbody>
</table>

In the majority of cases, the expert took the ‘directing’ role (Table G.19) - this supports the theory. In one case, the respondent reported that the newcomer was knowledgeable and so the director role was shared. Both of these support the theory that knowledge of the codebase is a prerequisite for effective directing.

Table G.19: Support for session configuration (director role)

<table>
<thead>
<tr>
<th>Usually:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>The expert guides the newcomer around the system.</td>
<td>14</td>
</tr>
<tr>
<td>The newcomer looks around the system and the expert provides commentary.</td>
<td>2</td>
</tr>
<tr>
<td>Other (equal directorship)</td>
<td>1</td>
</tr>
</tbody>
</table>
Finally, the theory states that the ‘driver’ role (i.e. control of the keyboard and mouse) is correlated with the ownership role, and that newcomers struggle to drive without a high-level knowledge of the system.

Unlike the observed sessions, where most driving was performed by the expert, respondents report newcomers in the driver role more often (Table G.20). However, the relationship between owner and driver is supported (Table G.21) - in the majority of sessions, either the owner is the driver, or has taken over as the driver by the end of the session. Given the strength of this relationship, this difference in the drivers between groups is consistent with the difference in owners, although it is not clear what causes the difference.

Table G.20: Support for session configuration (driver role)

<table>
<thead>
<tr>
<th>Usually, the mouse and keyboard are operated by:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>The newcomer</td>
<td>6</td>
</tr>
<tr>
<td>The expert to start with, then the newcomer</td>
<td>5</td>
</tr>
<tr>
<td>The expert</td>
<td>3</td>
</tr>
<tr>
<td>They each operate their own machine</td>
<td>2</td>
</tr>
<tr>
<td>Other</td>
<td>1</td>
</tr>
</tbody>
</table>

Table G.21: Support for relationship between owner and driver

<table>
<thead>
<tr>
<th>Usually, sessions are conducted on:</th>
<th>Responses</th>
<th>Same owner and driver by end of session</th>
</tr>
</thead>
<tbody>
<tr>
<td>The newcomer’s machine</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>The expert and the newcomer use their own machines, side by side.</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>The expert’s machine</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A third machine</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>“Expert machine via screensharing software”</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>“50/50 split”</td>
<td>1</td>
<td>unclear</td>
</tr>
<tr>
<td>“Not important”</td>
<td>1</td>
<td>unclear</td>
</tr>
<tr>
<td>“Initial run through on experts machine, more deep dive on newcomer’s machine”</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

One respondent reports using the newcomer’s machine in a different way:

Respondent, in questionnaire: [Usually, the mouse and
keyboard are operated by:] A mix; usually starts with the newcomer, but then to speed things up, when the developer wants to talk about module X or package Y, they will take over and navigate to it, then scroll up and down as they talk to show the parts they want to refer to.

This implies that the expert perceives that the newcomer’s navigation is too slow, but rather than resort to back-seat driving, the expert takes over. This supports the theory that newcomers have difficulty navigating quickly through the code and that experts find this frustrating, but is a previously unseen way to handle the problem.

**Session timing**

The theory describes session timing relative to two events: the newcomer achieving a working development environment, and the newcomer receiving their first development task. Sessions taking place before each of these events are linked to newcomer difficulties after the session.

The theory states that running sessions before the newcomer has a task is a problem, because without a task to provide context for the session, its content is often not relevant to the newcomer.

The information from respondents suggests a more nuanced interpretation: it is not enough for the newcomer to have a task before the session, because the expert may not use it for guidance (for example, because it is not within their area of expertise, as seen in S10.) Thus, the theory is amended to show that discussing the task within the session has a positive influence on the session’s relevance, and this requires specific knowledge on the part of the expert.

This amended theory is supported (Table G.22): of the twelve respondents who report that newcomers were given a task before or after the session, ten report that newcomers are unsure what to do afterwards, whereas all five respondents who reported that newcomers are given a task in the session do not report this problem.

One respondent gives an alternative view:

**Respondent, in questionnaire:** [The session timing] could be improved - I believe more time is necessary for the newcomer to understand the codebase before being given a task.
Table G.22: Support for session timing (relative to task)

<table>
<thead>
<tr>
<th>Sessions usually take place:</th>
<th>Responses</th>
<th>Newcomer unsure what to do after session</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the newcomer has been given a development task.</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>The newcomer is given a development task during the session.</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>After the newcomer has been given a development task.</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Whenever the expert is available (irrespective of the newcomer’s task).</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The theory also states that without an assigned task, newcomers are more likely to experience delays before working on the code, which reduces retention of session content. However, respondents who reported that sessions took place before tasks were assigned did not report these issues more than other respondents. This indicates that the link between a lack of a task and being diverted after the session is not as strong as it appeared from the observations, and the theory is modified accordingly.

In the theory, newcomers will have difficulty making use of the information from the sessions when they have not completed their development environment setup.

Most respondents report that sessions do not take place until the environment is set up, for this reason:

Respondent, *in questionnaire*: The sessions are well timed because the newcomer has the tools at their disposal to check things that are discussed during the walk through.

These responses (Table G.23) mostly support the theoretical link between session timing and development environment difficulties. However, given that most sessions took place on newcomer-owned machines, it is not clear how some sessions were able to proceed effectively without a working development environment. It is possible that those sessions did not include the full range of development activities (e.g. building, testing, running) and thus did not encounter the environment issues - no such issues were reported in the comments. Accordingly, the
theory is amended to make clear that an incomplete development environment is a problem for the newcomer after the session but not necessarily during the session.

Table G.23: Support for session timing (relative to environment setup)

<table>
<thead>
<tr>
<th>Usually, sessions take place:</th>
<th>Responses</th>
<th>Environment problems after session</th>
</tr>
</thead>
<tbody>
<tr>
<td>After the newcomer’s development environment is set up.</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>While the newcomer is still setting up their environment.</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Whenever the expert is available (irrespective of the newcomer’s environment)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Part of the session is to help the newcomer set up their environment.</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Development environment problems

In addition to describing the factors that influence whether the newcomer has development environment problems, the theory also describes newcomers’ difficulties in making use of sessions without a working development environment, and their struggle to complete the environment setup without expert assistance.

The responses (Tables G.24 and G.25) confirm that newcomers value expert help with their environment setup during the session. Remembering session content is also a problem, so the ability to consolidate knowledge immediately afterwards is important.

Table G.24: Support for development environment problems after the session

<table>
<thead>
<tr>
<th>After the session, it’s common for the newcomer to have problems with:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remembering the content of the session</td>
<td>8</td>
</tr>
<tr>
<td>Development environment setup</td>
<td>7</td>
</tr>
<tr>
<td>Missing test data</td>
<td>4</td>
</tr>
<tr>
<td>Problems running the software as an end user</td>
<td>3</td>
</tr>
</tbody>
</table>
APPENDIX G. ANALYSIS OF GENERALISATION QUESTIONNAIRE

Table G.25: Support for value of environment help

<table>
<thead>
<tr>
<th>For the newcomer, the most useful activities in a session are:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixing problems with the newcomer’s development environment</td>
<td>10</td>
</tr>
</tbody>
</table>

Session content

According to the theory, two important types of content for newcomers are a high-level overview of the system (which allows them to navigate the codebase) and a more detailed view of areas relevant to their task (which allows them to begin to contribute).

The responses (Tables G.26 to G.28) show strong support for the importance of the high-level overview, and moderate support for the importance of relevance to the newcomer’s task. The responses also indicate that low-level code representation has little value. Comments on this support the idea that, as experienced developers, newcomers do not need help with low-level implementation details, but benefit from high-level project-specific guidance.

Respondent, in questionnaire: It is reasonable to expect that programmers can deal with the programming side [...] The expert provides the context-specific information that could not be found elsewhere: the organisation of this code base [...]

Table G.26: Support for session content: during the session

<table>
<thead>
<tr>
<th>“During the session, it’s common for the newcomer to have problems with: (check all that apply)”</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigating the project</td>
<td>11</td>
</tr>
<tr>
<td>Unclear names in code</td>
<td>6</td>
</tr>
<tr>
<td>Information that is not relevant to their task</td>
<td>5</td>
</tr>
</tbody>
</table>

Expert knowledge and session preparation

In the theory, some experts prepare for sessions by reviewing or commenting the code to be explained, or by planning the session’s content. However, many do not, and this is associated with making mistakes in the explanation.
Table G.27: Support for session content: after the session

<table>
<thead>
<tr>
<th>After the session, it’s common for the newcomer to have problems with:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Being unsure what to do</td>
<td>10</td>
</tr>
<tr>
<td>Unclear names</td>
<td>6</td>
</tr>
<tr>
<td>Locating code</td>
<td>6</td>
</tr>
</tbody>
</table>

Table G.28: Support for session content: useful activities

<table>
<thead>
<tr>
<th>For the newcomer, the most useful activities in a session are:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting a high-level overview of the components of the system</td>
<td>11</td>
</tr>
<tr>
<td>Getting advice about the newcomer’s next development task</td>
<td>9</td>
</tr>
<tr>
<td>Going through complex code line-by-line</td>
<td>1</td>
</tr>
<tr>
<td>Going through all the code line-by-line</td>
<td>0</td>
</tr>
</tbody>
</table>

The responses (Table G.29) support the theory that most experts do not prepare for sessions. The most common reason for not preparing was the belief that, as authors of the code, they already had a comprehensive understanding of it.

Respondent, *in questionnaire*: No, I like to think I know my software.

Table G.29: Support for lack of preparation

<table>
<thead>
<tr>
<th>Before explaining code to a newcomer, do you do any preparation?</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>7</td>
</tr>
<tr>
<td>No answer</td>
<td>5</td>
</tr>
<tr>
<td>Yes (reviewing code)</td>
<td>3</td>
</tr>
<tr>
<td>Yes (other activity)</td>
<td>2</td>
</tr>
</tbody>
</table>

Like the original participants, experts who reviewed the code did so because they anticipated that their knowledge could be out of date, due to recent development work or their own forgetfulness.

Respondent, *in questionnaire*: I check the latest version of the code and make sure I understand the most recent changes.
Respondent, in questionnaire: Maybe reopen the code to remind me what the heck it is.

This highlights the effect of temporal issues on the expert, and this link is added to the theory.

The theory also reflects the issue of experts having forgotten the details of the code they are explaining. This is somewhat supported (Table G.30). Respondents report that forgetting code details is common in large codebases but not smaller ones:

Respondent, in questionnaire: [Agree, because:] The codebase is massive; we are bound to forget how some parts work until we look at it again.

Respondent, in questionnaire: [Disagree, because] Codebase is quite small.

Table G.30: Support for experts forgetting code details

<table>
<thead>
<tr>
<th>It’s common for the expert to be unfamiliar with some of the code.</th>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

However, in the observed sessions, code details are forgotten in codebases of all sizes. One possible explanation is that code is forgotten when a developer has not interacted with it for some time, and this scenario is more common with large codebases but can also occur when a small codebase is no longer being maintained. It is also possible that many experts do not expect this to be an issue (as evidenced by the lack of preparation as well as their survey responses) but in practice it is common (as observed in sessions.) However, investigation of this issue remains for future work. While the cause is unclear, there is some support for the issue of experts forgetting code.

Helping newcomers with development environment setup

The theory also describes experts struggling to help newcomers with their development environment setup because they have forgotten the details of
APPENDIX G. ANALYSIS OF GENERALISATION QUESTIONNAIRE

this one-time operation. This aspect of theory is not supported by the respondents (Table G.31), but the comments indicate a more complex picture. Respondents report that projects with simple environments are not affected by this issue, while more complex environments are automated to mitigate this problem.

Table G.31: Support for experts forgetting setup details

<table>
<thead>
<tr>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neither agree nor disagree</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Respondent, in questionnaire: [Disagree, because:] Most of the process is automated so there is little room for mistakes. Experts do occasionally forget some of the details.

Respondent, in questionnaire: [Strongly disagree, because:] We have a very simple environment.

Respondent, in questionnaire: [Strongly disagree, because:] All developers at all levels [are] expected to be current and conversant with development processes and environments.

The latter quote does not discuss whether this ideal is met in practice. It may be addressing a different problem - as non-novices, both the expert and the newcomer would indeed be expected to be competent users of IDEs and process tools, but this is not the same issue as the project-specific difficulties with environment setup that were seen in the observed sessions and interviews.

Without more information, particularly details of the various environments, it is not possible to be sure of a link between environment complexity and difficulties with setup.

A minority of respondents have encountered this issue, and one reports on an attempt to mitigate it:

Respondent, in questionnaire: [Strongly agree, because:] This happens very frequently. We built a tutorial for helping them, which was helpful, but they also often forget to read it.
This aspect of the theory is removed, and left for future investigation.

**Project terminology**

The theory states that project terminology is initially problematic for both newcomer and expert. Without this shared vocabulary, newcomers struggle to express questions and experts have difficulty making their information understood. The responses (Tables G.32 and G.33) support the theory.

<table>
<thead>
<tr>
<th>Table G.32: Support for project terminology issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>“During the session, it’s common for the newcomer to have problems with: (check all that apply)”</td>
</tr>
<tr>
<td>The project terminology</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table G.33: Support for value of help with terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the newcomer, the most useful activities in a session are:</td>
</tr>
<tr>
<td>Learning the project terminology</td>
</tr>
</tbody>
</table>

**Information overload**

The theory describes how, even with preparation, some experts felt under pressure to fill the session with as much information as possible, while newcomers struggled with information overload.

The responses (Tables G.34 and G.35) provide strong support for this theory. In particular, too much information was the most frequently reported problem, while too little information is much less common. Similarly, there is support for the problem of experts going too fast through the system, but no respondents reported problems with experts going too slowly. The irrelevant information is sometimes a problem for newcomers, and remembering all the content was a problem for about half of respondents.

In addition, some comments describe information overload issues:

**Respondent, in questionnaire:** [Newcomers have difficulty with:] Being interested in understanding the whole system,
APPENDIX G. ANALYSIS OF GENERALISATION QUESTIONNAIRE

Table G.34: Support for information overload issue

<table>
<thead>
<tr>
<th>“During the session, it’s common for the newcomer to have problems with: (check all that apply)”</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Too much information</td>
<td>14</td>
</tr>
<tr>
<td>The expert going too fast through the system</td>
<td>9</td>
</tr>
<tr>
<td>Information that is not relevant to their task</td>
<td>5</td>
</tr>
<tr>
<td>Not enough information</td>
<td>4</td>
</tr>
<tr>
<td>The expert going too slowly through the system</td>
<td>0</td>
</tr>
</tbody>
</table>

Table G.35: Support for information overload effects

<table>
<thead>
<tr>
<th>After the session, it’s common for the newcomer to have problems with:</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remembering the content of the session</td>
<td>8</td>
</tr>
</tbody>
</table>

rather than focus on their particular area/project, then having information overload or confusion of system interactions.

Respondent, *in questionnaire:* [On improving onboarding sessions:] One big upfront code talk sounds good initially, but realistically the newcomer’s eyes glaze over and very little sinks in; they just come back to ask more questions later on. Although giving very little detail upfront seems perhaps unfriendly (or like you are just letting them flounder), I think it works well to have the newcomer ask questions as they need to.

This aspect of the theory is strongly supported.