



Characterizing the transfer of program comprehension in onboarding: an information-push perspective

Rebecca Yolande Yates, NORAH POWER, JIM BUCKLEY

Publication date

01-01-2019

Published in

Empirical Software Engineering;25, pp. 940-955

Licence

This work is made available under the [CC BY-NC-SA 1.0](#) licence and should only be used in accordance with that licence. For more information on the specific terms, consult the repository record for this item.

Document Version

1

Citation for this work (HarvardUL)

Yates, R.Y., POWER, N.and BUCKLEY, J. (2019) 'Characterizing the transfer of program comprehension in onboarding: an information-push perspective', available: <https://hdl.handle.net/10344/9834> [accessed 23 Jul 2022].

This work was downloaded from the University of Limerick research repository.

For more information on this work, the University of Limerick research repository or to report an issue, you can contact the repository administrators at ir@ul.ie. If you feel that this work breaches copyright, please provide details and we will remove access to the work immediately while we investigate your claim.



Characterizing the transfer of program comprehension in onboarding: an information-push perspective

Rebecca Yates¹ · Norah Power¹ · Jim Buckley¹

Published online: 26 July 2019

© The Author(s) 2019, corrected publication 2021

Abstract

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 share and transfer understanding of their codebases. This is common during the onboarding process, when a new developer has joined a project or a company. The work reported here uses a Grounded Theory approach to explore the different types of information passed from experts to newcomers during onboarding, and the perceived value of these types. The theory is grounded in field-study data collected during twelve in-situ onboarding sessions, across eight organizations, with a design based on two pilot studies that were carried out in advance. The field-study data was supplemented and validated with interviews and questionnaires. It provides a description of four views through which the experts represent their code to the newcomers, revealing several interesting aspects of expert-led program comprehension. In particular, it provides evidence that extends current thinking on the temporal aspect of code: where experts discuss changes that have been made to the code-base, changes that are currently being made to the code-base (including temporary fixes) and changes intended for the code-base in the future. In addition, a rationale-based view of the code-base is emphasized in the findings, making explicit the system's functional/non-functional requirements, and their impact on the system's design. This information was perceived as highly valued by the newcomers. Additionally, Structural and Algorithmic views, which have already been firmly established in program comprehension literature, were also noted in these onboarding sessions.

Keywords Program comprehension · Information seeking · In-situ onboarding · Grounded theory

Communicated by: Emerson Murphy-Hill

✉ Rebecca Yates
rebecca.yates@fastmail.com

Norah Power
norah.power@ul.ie

Jim Buckley
jim.buckley@ul.ie

¹ Lero, University of Limerick, Limerick, Ireland

1 Introduction

The majority of software developers report that understanding another's code is a serious problem (LaToza et al. 2006). This difficulty affects 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 and evolve high quality software on time.

Attempting to understand unfamiliar code is, at its core, a problem of program comprehension. Seminal empirical studies of program comprehension (e.g. Rist 1986; Pennington 1987; Detienne and Soloway 1990) have resulted in evidence for various theories. These theories have mainly focused on the information programmers attempt to extract from unfamiliar code and how it is obtained. This work has led to the development of program comprehension models for unguided investigations (for example von Mayrhauser et al. 1997; Detienne 2002), which model the developers' approaches for matching their existing, general knowledge of programming and a system's domain to the specific goals of unfamiliar code.

A more applied view of program comprehension is to consider it as either information seeking (O'Brien 2007) or as an exercise in concept and feature location (Dit et al. 2011). Information seeking research has provided models for locating, understanding 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 pervasive in industrial software development/evolution. Within this perspective, for example, several groups of researchers (Ko et al. 2006; Lawrance et al. 2013; Ragavan et al. 2016) have probed an information foraging model in source code exploration, where software information is the prey and the maintainer the forager. In contrast, concept and feature location research has concentrated more on automated approaches to support developers seeking existing aspects or functionality in unfamiliar source code as part of development/evolutionary work (Razzaq et al. 2019; Chochlov et al. 2017).

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 other studies reported in this area, this scenario has a key difference: In the previous studies of program comprehension, information seeking, and concept location, the developers were attempting to pull the required information from the passive materials available. This is typically a slow and frustrating process because the developer, working in an unfamiliar landscape, is less able to judge what is important and where to find it (Sim and Holt 1998; Dagenais et al. 2010). But when they are not left to themselves, a knowledgeable colleague can push information synchronously to the developer (Begel and Simon 2008b) and, based on their past experience, this colleague is in a much better position to identify and locate relevant information than someone who is new to the code.

In this paper, program comprehension is studied from this alternative perspective, as contextualized in Fig. 1. (Terms such as *Newcomer*, and *Expert*, used in the figure have specific meanings for this work and are defined in Section 2 below). Instead of starting from a position of *Information pull*, as shown at the top of the figure, the research reported here is focused on the *Onboarding session*, towards the bottom of the figure. In this session, the *Expert* describes the unfamiliar codebase to the *Newcomer* in a more *Information push* context. The content of such sessions can be explored to answer the question:

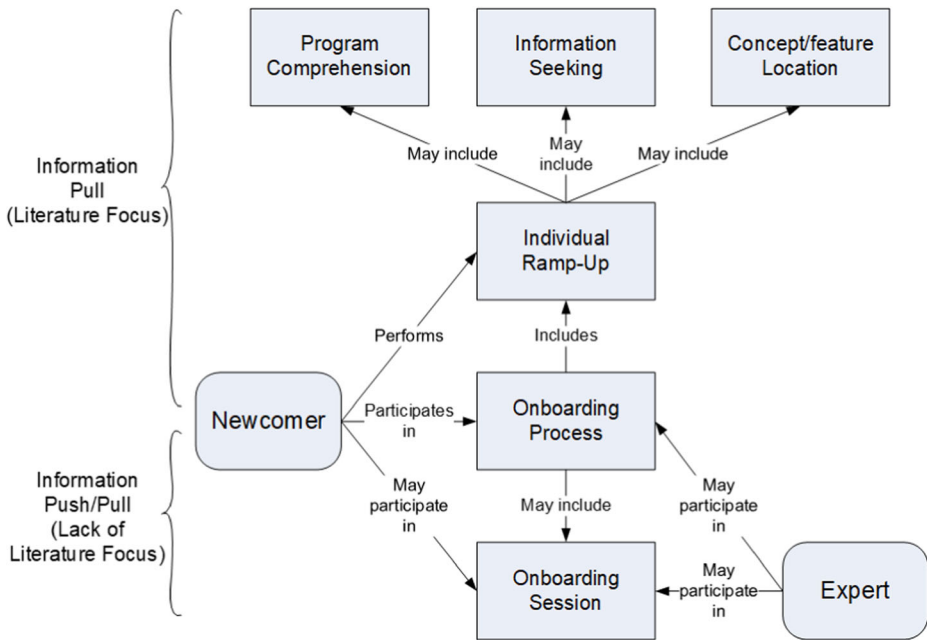


Fig. 1 A context map for this research

What information do experts usefully present when transferring their system-knowledge to newcomers during onboarding?

Once derived, the answer to this question reveals important clues as to the core characteristics of the information passed from expert to newcomer: the information types that experts consider developers may need when faced with a new, complex, software system (push information). In addition, because such sessions are interactive, information may often be offered by the expert in response to a newcomer's questions (information-pull). The degree to which that occurs is also discussed.

In the work reported here, the knowledge transfer between the expert and a newcomer starting on an unfamiliar, yet substantial, software system is captured primarily through observing and recording in-situ onboarding sessions, in which experts talk newcomers through the codebase. These sessions occur as part of the onboarding process (along with other onboarding activities) in commercial (Johnson and Senges 2010) and open source software systems (Fagerholm et al. 2013). Using the terminology introduced by Van Maanen and Schein (1979), it can be described as joint expert and newcomer (collective) activity, and is more or less segregated from other regular team members. Johnson and Senges (2010) would further categorize this as face-to-face (off-line) training, focusing on the real code base (codewalks), but with the exception that they do prepare developers to perform ‘in the wild’.

The next section contains a taxonomy of terms used throughout the paper. Section 3 reviews the key theories of program comprehension and programmer information seeking. Section 4 contains the study design followed by the results in Section 5. These results are discussed in Section 6, with the conclusions presented in Section 7.

2 Taxonomy of Terms

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

Newcomers are skilled software developers who lack specific knowledge of the code they will work on. The term is used to distinguish these developers from novices, who do not have significant development skill. In other studies of onboarding, skilled developers in this position are also known as *apprentices*, *expert apprentices* (Berlin 1993), *new hires* or *software immigrants* (Sim and Holt 1998).

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 thereof. They are experts in the context of this system, the dependencies used to implement the system (e.g, libraries, frameworks, APIs) and the application domain (as opposed to newcomers). Other names for experts include *team historians* (LaToza et al. 2006), *consultants* (Berlin and Jeffries 1992) and *mentors* or *veterans* (Sim and Holt 1998).

Onboarding is a process by which newcomers learn to work in their new team. In this study, 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).

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).

Push and *pull* are terms used in this study to nuance between guided and unguided program exploration. *Pulling* information refers to extracting information 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 3.1). 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 actively providing information about code to a newcomer, which is more likely to be prevalent in guided program exploration, when guided by a developer more experienced on the system. Studies of professional developers at work reveal a culture of mentoring, in which expert developers pass on technical (and other) information to help newcomers familiarize themselves with the code. Push and pull as concepts are developed further in Section 3.3.

3 Program Comprehension and Information Seeking

In this section existing theories and perspectives on program comprehension and information seeking are presented. In Section 3.1, program comprehension research is summarized as a basis for discussing the findings generated from this study. Notably, the discussion (please see Section 6) shows that several of these theories are supported, expanded on or challenged, based on the findings from this study.

The study presented here identifies the types of information presented by experts and valued by novices. As such it relates directly to the existing research on programmer information seeking, particularly with respect to the information types sought, but also with respect to the information sources consulted and the information foraging behavior exhibited by programmers. Information seeking and information foraging theories are presented in section 3.2.

Finally, section 3.3 discusses how most of the work in these fields has adopted the perspective of developers pulling information from static representations like source code. As this paper is centred around an activity where there is the opportunity for experts to push information to newcomers, section 3.3 also identifies several research works where developers relied on colleagues to push appropriate information towards them, either in documentation, or in real-time, proactively or reactively in response to questions.

3.1 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 characterized by Boehm-Davis et al. (1996) as “reconstructing the logic, structure and goals that were used to write a computer program”.

Detienne (2002) identifies three approaches to program comprehension in her review. The first approach is to consider program comprehension as the application of pre-existing knowledge schemas. In a programming context, these schemas correspond to programming plans, and recognition of plans can be triggered by certain beacons in the code (see the following paragraph). The second approach is one of constructing networks of relations based on the structure of the code. Thirdly, program comprehension can be approached as constructing a representation of the system in the context of the situation (application-domain, performance or other required attribute) within which it exists.

In terms of Detienne’s first approach, *beacons*, according to Brooks (1983), are sets of constructs in program code that indicate certain 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 scan for beacons to support their hypotheses about the code’s workings. Expert recognition of beacons was demonstrated by Wiedenbeck (1986). She showed that more experienced programmers recalled key lines (beacons) of procedures, while novices recalled only simple structural statements, suggesting a more linear reading of the code.

Building on beacons, a *plan* is a standardized 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, at a larger scale of granularity, the readfile-analyse-report program structure common in Information Systems. Soloway and Ehrlich (1984) showed that more experienced programmers could identify conventionally-written

plans in unfamiliar code, and hence could understand conventionally-written code more easily than novices who lacked a repertoire of plans. The operational nature of the plan-examples presented in this section suggests support for Detienne's situation approach (Detienne 2002).

The comprehension approaches used by programmers can be broadly categorized into top-down and bottom-up approaches. Bottom-up comprehension is a methodical process of understanding code line-by-line, and then chunking lines together into plans to build up understanding (as described by Shneiderman and Mayer 1979; Pennington 1987) without leveraging beacons. In contrast, when using a top-down strategy, a programmer begins with either hypotheses about the likely functionality of the code (Brooks 1983) or scans the code for beacons (Soloway and Ehrlich 1984), to confirm the presence of plans (O'Brien et al. 2004).

Von Mayrhauser and Vans (1995a) developed an integrated meta-model of program comprehension, arguing that programmers do not rely on one comprehension process exclusively (von Mayrhauser and Vans 1993, 1995b). Their model is based on four earlier models: Shneiderman and Mayer's (1979) bottom-up, chunking-code process, Brooks's (1983) expectation-driven process, Soloway and Ehrlich's (1984) inference-driven process and Pennington's (1987) detailed, bottom-up, chunking, networks-of-relations processes. Letovsky's (1987) observation that programmers were opportunistic processors, switching between top-down processes and bottom-up processes based on cues, was employed to describe how the models were integrated during comprehension. Lakhota's (1993) introspection report went further, suggesting that domain knowledge affects the predominance of strategy: more domain knowledge permits a more top-down, hypothesis-driven approach. O'Brien et al. (2005) empirically assessed this claim, in work that extended Shaft and Vessy's work (1995). O'Brien et al. 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 types of domain, but in a familiar domain it became a secondary strategy used to investigate the details of code once the overall purpose was understood.

Also related to the application of these models is Detienne's observation that program comprehension is influenced by the task context. An illustrative example is provided by Pennington (1987). Here, professional programmers were asked questions about a short code snippet to learn whether their mental representation corresponded to the structure of the code or whether it was based on the intent of the code. The result was strongly in favour of a code-structure representation, constructed bottom-up. However, the task given to the programmers in this instance was simply to understand and recall the 15-line program. A later experiment by Pennington, reported on in the same paper (1987), using a 200-line program and a modification task, showed that the dominant mental representation changed from an initial code-structure based representation to a more functional representation after the modification task had been completed.

Closer to the work presented here is collaborative program comprehension. Storey (2006) acknowledges that pair programming is an important direction for program comprehension and laments the fact that there is little research into this approach. Van Deursen (2001) points out that paired programming forces programmers to think-aloud, making comprehension strategy explicit thus helping the partner, and that the combined strengths of the pair help them reach a better software understanding. In terms of new hires, Poff (2003) found that paired programmers acquired increased environment and technical knowledge quicker, but he did not focus on the comprehension process or the categories of information acquired beyond these. Sillito et al. (2006) focused on the questions newcomers ask during comprehension,

using an empirical methodology where pair programmers' utterances were recorded as they performed software change tasks and the transcripts searched for questions. They categorized questions into those aimed at specific points of focus, questions building on those points, questions relating those points to other points (sub-graphs) and questions over multiple sub-graphs of 'point-relations'.

3.2 Programmer 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, Goncalves et al. (2011) observed software developers at work and found that information seeking activities took up on average 32% of their time, demonstrating that information is sought for a wide range of software development activities, including comprehension. Research work on programmers' information seeking can be divided into the study of information sources, information needs and information-seeking models.

With respect to information sources, source code is still considered the most reliable information source on a software system (Singer 1998; Seaman 2002; Sillito et al. 2008) and studies tend to focus on that artefact (McKeogh and Exton 2004). However, co-workers (Begel and Simon 2008a) and documentation are also considered valuable resources, albeit with some disadvantages. For example, it can be difficult to identify knowledgeable co-workers (Mockus and Herbsleb 2002) and, even if a knowledgeable co-worker is identified, poor explanation skills and unhelpful attitudes can impact on their value (McDonald and Ackerman 1998; Begel and Simon 2008b). Likewise, documentation might not exist, be out of date (Lethbridge et al. 2003), or be difficult to navigate through to information of interest (Dekel and Herbsleb 2009a; Sharif 2012).

Ko et al. (2007) categorized programmers' information needs into 21 different types, ranging from highly social (what are my co-workers doing?) to highly technical (how do I use this data-structure?). They found that historical information was important because it allowed developers to understand why the code was designed in a particular way. However, the why question (why has the code been implemented in this way?) was considered by the 17 developer participants in the study as the most difficult to answer. That finding aligns with Sharif et al.'s (2015) findings when studying the questions asked by Open Source programmers on mailing lists: they often ask for design-rationale information.

With respect to models of programmer information seeking, O'Brien (2007) combined Ellis and Haugan's (1997) model of scientists' information seeking with Wilson's (1981) discussion of information needs, Kulthau's model (1988) based on students using library resources, and Marchionini's (1997) model emphasizing electronic environments (which differed from more traditional environments in terms of the volume, format and accessibility of information). The resultant, proposed, programmer information-seeking model was tested and refined using think-aloud data from eight sessions with six software engineers conducting maintenance tasks. This resulted in a non-linear, iterative, six-stage model for the information seeking process of these software engineers:

- Becoming aware of the problem;
- Refining their understanding of the nature of the problem;
- Collecting information related to the solution;
- Examining and analysing the results;

- Information-prompted, information-seeking actions;
- Problem resolution.

Information foraging is another model of information seeking that has been applied to software developers (Pirolli and Card 1999; Lawrance et al. 2010, 2013; Ko et al. 2006). Information foraging makes an analogy between a predator seeking prey and the information-seeker, seeking information. The source code and documentation provide cues that give a ‘scent’ to the prey and the source-code corresponds to the topology that must be navigated when hunting. Early work in this area assumed a fixed prey. However, empirical studies suggested that, during software maintenance, the goal changes rapidly as new information is discovered, much like the O’Brien model above. The authors observed developers being ‘distracted’ by a new scent or hypothesis to investigate before fully investigating an old one. Lawrance et al. (2010) 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.

3.2.1 Feature Location

Closely related to program comprehension and information seeking, is the field of feature location. In source code, a feature refers to a user-functionality that a software system provides (Dit et al. 2011). Feature location then is the mapping of a feature to the source code entities that implement that feature (Chochlov et al. 2017). This is a non-trivial, information-seeking endeavour because code can be large/complex and its structure may not be aligned with its functionality. For example, a single functionality may be spread over the UI, business-logic and data-access layers of a system. It is, however, an important information-seeking endeavour because there are many software-evolution activities dependent on feature location. For example, users may report faulty functionality and it is the developer’s job to find the (faulty) code associated with that functionality.

Given the importance of feature location in software evolution, there have been numerous automated approaches suggested (Dit et al. 2011; Rubin and Chechik 2013). These approaches break into four categories: Dynamic, based on logged, execution traces where various system functionalities are exercised or not; Structural, based on leveraging the structural relationships (method invocations, variable accesses) in source code; Textual, where meaningful lexicons in the source code (variable names, method names, class names) are compared to developer queries based on natural language parsing, information retrieval techniques or simple pattern matching; and Hybrids of these three categories.

Of more relevance here are the few studies which look at feature location as performed manually, possibly with support. Starke et al. (2009) showed that simple pattern matching was ineffective as a developer search tool, whereas Jordan et al. (2015) showed that system experts were more effective at using such an approach to locate features in code. Buckley et al. (2016) found that experts tended to find a feature-foothold in the code very quickly, without specialist support, using simple pattern matching or their knowledge of the system’s package structure. They then tended to search concentrically out to discover the full extent of the feature in the code using structural relationships from that foothold. This finding was reflected by Chen and Rajlich (2011) in their work observing feature location when a developer was presented with a system dependency graph.

3.3 Information Pull and Information Push

The empirical work and theories described above reflect the majority of work in program comprehension and information seeking in presuming an entirely information-pull scenario, often where there is an implicit assumption that developers focus entirely on the code-base (von Mayrhauser and Vans 1995a; Kelly and Buckley 2006). However, individual exploration is only one aspect of professional program comprehension. Developers also explain their code to one another, particularly during the onboarding process (Sim and Holt 1998).

For example, Begel and Simon (2008a) found that, when debugging, the new hires at Microsoft needed colleagues to help them understand the code-base. More specifically, Sim and Holt (1998) identified the problems of coding standards and the ‘intricacies’ of the software system for newcomers. These findings are echoed by Perlow (1999) who found that software engineers could not complete many of their tasks without interacting with their peers. Berlin (1993) and Berlin and Jeffries (1992) probed the types of information provided by mentors to newcomers in such situations. They identified ‘difficult to find’ information and design rationale information, noting that the newcomer-mentor conversations tended to be highly interactive. However, they also found that the newcomers tended to be overloaded with information, a finding that aligns with Begel and Simon’s (2008b) findings: they stated that impromptu teaching sessions between more experienced colleagues and newcomers were not well structured in terms of the newcomers’ knowledge.

Murray and Lethbridge (2005b) touched on experts’ pushing of information by asking expert developers to explain their code to them “as if to a newcomer”, by drawing diagrams on a whiteboard. However, the study aimed to investigate low-level comprehension activity on the part of the expert, rather than the transfer of information to another developer and did so in an artificial context. The sessions were analysed for patterns of comprehension, described in more detail in Murray and Lethbridge (2005a).

Patterns included baseline landmarks (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 paper.

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. This is plausibly useful, but no evaluation of this idea was carried out.

Some other experiments contribute by chance to the investigation of information push in program comprehension. Lee and Kang (2012) conducted a Wizard-of-Oz study to examine the effects of a proposed code recommender system. A Wizard-of-Oz study involves participants who believe they are interacting with an automatic computer system but in actuality are interacting with a hidden expert who (partially) operates that computer system. Although not the purpose of the research, Lee and Kang (2012) can be framed as ‘expert push’ in that an appropriate visualization was pushed to the participant by this hidden expert as they navigated an unfamiliar system. The majority of participants found that the diagrams offered were useful

guides, with one participant repeatedly asking how the software was able to generate such a useful diagram.

Similarly, studies of documentation and commenting can provide some insight into expert 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 excerpts available through tooltips. In their study, 26 student participants performed significantly better on three debugging tasks (for which a directive was relevant) when eMoose was available.

Information can be pushed through the code itself, particularly through 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 the trade-offs that affected implementation.

However, that work was directed at asynchronous, passively-pushed information only and our review suggests that the vast majority of the empirical studies/theories carried out in this area have investigated an exclusively ‘information-pull’ situation. There is little evidence on the nature of the information exchange when information can be pushed to the newcomer, or how ‘information-push’ enabled scenarios can augment program comprehension. This paper addresses this through the study of in-situ, synchronous on-boarding sessions.

4 Study Design

The review in Section 3 suggests that there currently exists a lack of theory with respect to synchronous, pushed information during program comprehension/developer information seeking. Thus, in this research (where the aim is to derive theory with limited literature support) an inductive approach is desirable. In inductive research, insight is derived from empirical data (Buckley et al. 2006). Given that the phenomena under study are social, based on information communication, the research data will largely be verbal, textual or image oriented. Thus, the epistemology of the research presented here is more interpretivist (Stol et al. 2016), keeping the researcher grounded in the ‘lived concepts of everyday life’ (Denzin 1983). But ultimately the approach adopted is paradigm co-existence (de Gialdino 2009) and thus pragmatic (Easterbrook et al. 2008) towards the research goal, where interpretations are probed in a more quantitative fashion, to establish the strength or prevalence of the findings.

Primarily this study focuses on inductive, qualitative data analysis of a complex phenomenon (Seaman 1999), in that it probes human behaviour (communication) and characteristics. Here, Grounded Theory (Glaser and Strauss 1967) is adopted as a systematic and accepted method for generating new theory from data. However, the approach adopted is more aligned to a Constructivist Grounded Theory perspective where data/theories are constructed by the researcher as a result of their interactions with the field and participants (Charmaz 2009). Most

notably, in this theoretical framework, a literature review is employed without forcing that review on the grounded data. Illustrative examples of Grounded Theory's adoption in the software engineering sphere include Dagenais et al.'s (2010) description of the onboarding process and Hoda et al. (2012) research on self-organizing agile teams.

The overall design of the study is illustrated in Fig. 2. Two small scale-pilot studies were initially carried out to refine the data-collection methodology for the empirical study. Then twelve onboarding sessions were observed and recorded, and the data transcribed from video captures. Open coding was then performed on the data, where recurrent and distinctive concepts were identified, memoed and related through immersive study and constant comparison. This open coding led to axial coding: the identification of more abstract categories around these codes, again via constant comparison. The codes, their characteristics and the encompassing codes formed an initial theory of (in this case) onboarding information exchange. This theory was evaluated, through the collection and analysis of more onboarding-session data and through a questionnaire, towards a revised, refined theory.

Qualitative approaches (Lethbridge et al. 2005) such as Grounded Theory encourage the interleaving of data collection with the analysis during the study. As per Lethbridge's assertion, and in line with the schema in Fig. 2, this research also interleaves data collection with analysis, allowing the collection of data appropriate to the emerging theory. It also allows

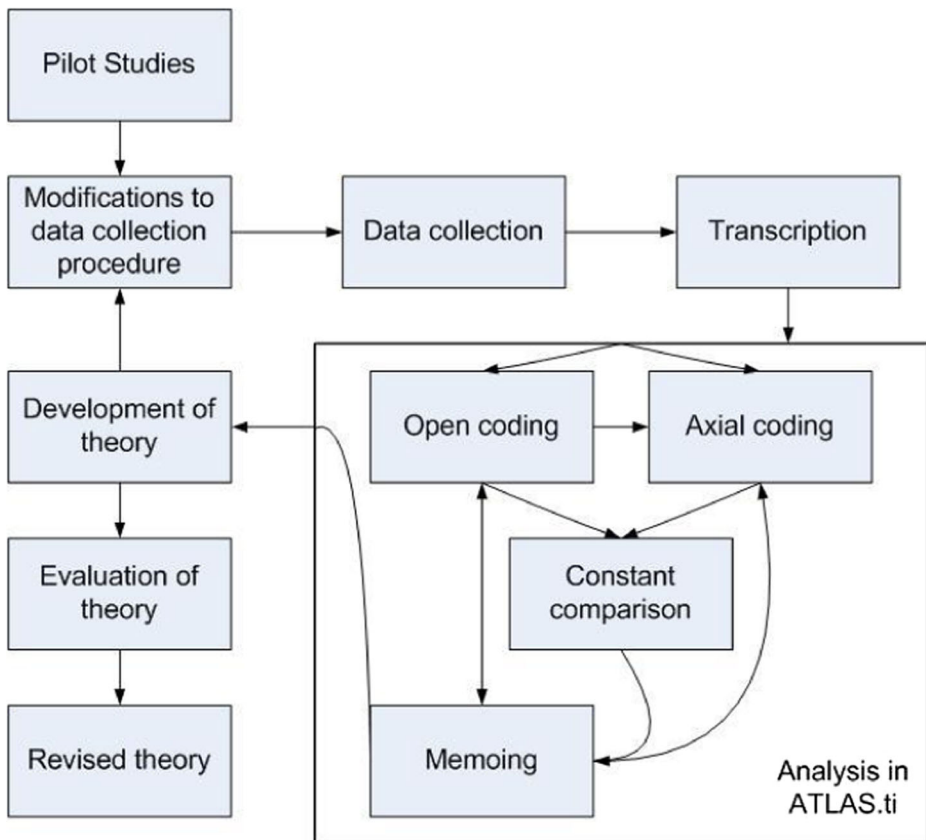


Fig. 2 The overall design of the study

selected data collection techniques to be refined and tailored while the study is in progress. Such a refinement process was necessary, as well as valuable, as there were no previous studies of information exchange in onboarding sessions that could be used as models and because the study of onboarding presents a particular challenge for the researcher: specifically, finding suitable research subjects/participants. Onboarding is a widespread occurrence in software engineering but it tends to happen sporadically, and at short notice, thus impeding efforts to locate suitable companies that were both willing and in a position to participate in the field study of this phenomenon in a timely way, given the additional complication of drawing up of acceptable nondisclosure agreements. Participants for the field studies were sought widely and through a variety of means including social media and personal contacts. However, the expectation of having several in-situ onboarding sessions happening at roughly the same time was not realistic.

4.1 The Pilot Studies

Data collection began with two pilot study sessions in which aspects of the research design were tested, thereby avoiding costly mistakes. Neither session yielded data that was used in the final analysis. The first one tested the data capture techniques that would be used in onboarding sessions. The researcher played the role of the newcomer while an expert explained a software system using the screen, a whiteboard and pen-and-paper diagrams. This revealed the need for both screen recording software and a movable camcorder to capture the discussions. The camcorder would not provide a readable view of the screen, but was needed to capture the conversation and the activity in the session away from the computer.

The second pilot again involved the researcher in the role of the newcomer and highlighted the unsatisfactory nature of using the researcher in this role (which had been conceived as a possible solution to the problem of finding enough participants, based on Murray and Lethbridge's design (2005b)). Even though it was agreed that the researcher/newcomer would make some small modification to the software (to add realism to the task), it became clear at this stage that the researcher playing the role of newcomer in each session was not going to give realistic data. This protocol was therefore abandoned in favour of more naturalistic field studies.

4.2 Data Collection

Ultimately, the data collected in this study are drawn from twelve onboarding sessions across eight different organizations. In these sessions, data collection consisted of three main tasks for each onboarding session: recording the session, participant questionnaires, and follow-up interviews. In terms of recording the session, a camcorder recorded the session from the newcomer's point of view. The camcorder was small enough to be minimally intrusive and could be re-positioned by the researcher to record off-screen activity such as diagram sketching; otherwise, the researcher had no interaction with the participants during the session. The content of the screen displaying the software that was being introduced 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. Thus, two parallel video recordings were made in each session, one focusing on the software, the other on the participants and their interactions with each other and with the computer displaying the software of interest.

After each session, the participants were asked to fill out an online questionnaire about their backgrounds as software engineers. It was important to establish that the participants in this study, both experts and newcomers, were distinguished from novices by their skill in software development, and this was done informally in advance of each session. However, the questionnaire checked their level of skill in order to provide corroborating evidence in this regard. Accordingly, three measures were used to approximate skill: experience (in years), self-reported programming expertise and the number of programming languages they know. The latter two measure have been demonstrated to correlate well with skill, for less experienced developers (Sheppard et al. 1979; Feigenspan et al. 2012), with years spent programming also having been shown to have a moderate correlation. The questionnaire also asked some open-ended questions about their experiences of onboarding sessions. To prevent these questions from influencing the session, the questionnaire was presented afterwards in each case.

The questions were reviewed for minor refinements 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, one answer to the question ‘how long have you worked here?’ caused difficulty for a participant who had worked at the company as a contractor, then worked elsewhere, and then returned as an employee; important details that would be lost in a purely numerical answer. (The final iteration of the questionnaire is available in Appendix 1).

After completing two sessions, it became clear that the sessions themselves provided very little information on whether the newcomer had found the exercise useful; that is, whether effective information transfer was taking place. To address this, from the third session onwards, the newcomers were interviewed about the effectiveness of the session. To permit an informed response to this question, the interviews were conducted between two-to-four months later, after the newcomers had attempted to modify the code under discussion. 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 each session. To avoid bias, the standard questions were always asked first. Leading questions were avoided. This sequenced, 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 2. As noted by Corbin and Strauss (2008, p. 28), the participants often volunteered more information after their interview had formally ended. This included further reflections on the onboarding session, demonstrations of tools, and example diagrams. The sessions resulted in twelve sets of video, nine interviews and 24 questionnaires (from the newcomers and the experts). In total, they involved 23 developers and two other participants.

Many of the findings were based directly on frequent observations from the session data or on common statements made by the participants. Others were more indirectly interpreted from the events of the sessions. The indirectness of these latter findings needed to be tested for ‘fit’ by the participants (Corbin and Strauss 2008). Hence, subsequent to data analysis, these findings formed the basis for a list of statements that were given to participants and they were asked to rank their agreement to each of these statements on a Likert scale. They were also given a comment box for each statement to allow more nuanced responses, especially in the case of strong opinions or neutral answers. It should be noted that while many of the statements were phrased to be in line with the findings, several were phrased in conflict with the findings to deter bias. The full questionnaire can be found in Appendix 3.

4.3 Session Characterization

Table 1 describes the twelve sessions in terms of the systems under study. As can be seen, the sessions covered a wide range of technologies and application domains. The systems ranged in size from 1KLOC to 28KLOC and from 24 to 587 files. On average, they were 9.8 KLOC long and contained 118 files. Table 2 then characterizes the sessions in terms of the newcomer characteristics and tasks. As can be seen from this table, two sessions (1 and 12) involved more than one newcomer, and in other sessions, extra persons were in attendance. In addition, the time the newcomer had the opportunity to get familiar with the system before the session varied from hours to seven months, but where newcomers were with the organizations for more than one month, the sessions concentrated on parts of the system unfamiliar to them. Sessions 7, 9 and 12 happened in the same organization and with one of the same newcomers. Sessions 4 and 5 and sessions 10 and 11 did likewise.

The experts rated themselves more familiar with the code base (4.5/5 on average, standard deviation 0.5, minimum 4), but otherwise were similar to the newcomers in profile: they reported knowing 5.7 other programming languages (SD 2.4, min 2), reported their programming expertise as 4.5/5 (SD 0.5, min 4), but had a slightly higher level of experience, on average: 14.2 years (SD 6.8, min 5.5). With regard to commercial experience they were largely similar to the newcomers (6 years, SD 4.7, min 0).

The newcomers had, on average 10.3 years of software development experience (SD 7.6, min 2.25) of which, on average, 5.2 (SD 7.5, min 0) was commercial experience. Their (self-reported) programmer expertise was 4.3/5 on average (SD 0.5, min 4) and they knew, on average, 5.9 programming languages (SD 2.2, min 3). They said they were moderately familiar with the code (3.1/5 on average, SD 1.0, min 2) but this mark reflected their knowledge of the

Table 1 System characteristics for the sessions

Sess.	Software		Language/ Technology
	Commercial/Academic	Application Domain	
1	Academic	Open source visualization system responsible for visualizing source code, at various levels of abstraction, primarily graphically.	Java
2	Commercial	An in-house, inventory management system for a wholesale office-products supplier	C#, SQL
3	Commercial	A Toll-enforcement system (notification, requests for payment, etc.)	C#, SQL
4	Commercial	A system for reporting on various structural software metrics of software products to management	Javascript
5	Commercial	A system for reporting on various structural software metrics of software products to management	Python
6	Academic	A prototype network-analysis system, that was implementing the novel analysis techniques derived by the group	Java, Javascript
7	Academic	Software re-engineering tool directed at architecture conformance and recovery through source code analysis (Buckley et al. 2013).	Java
8	Commercial	A web application for management of telephone calls within large companies.	Python, Javascript
9	Academic	Software re engineering tool directed at architecture conformance and recovery through source code analysis (Buckley et al. 2013).	Java
10	Commercial	Component infrastructure/framework development for in-house usage.	Java, Flex
11	Commercial	Component infrastructure/framework development for in house usage.	Java, Flex
12	Academic	Software re engineering tool directed at architecture conformance and recovery through source code analysis (Buckley et al. 2013).	Java

Table 2 Newcomer characteristics and tasks for the sessions

Sess. Newcomer				Notes
No.	Pre-exposure to System	Context	Present	
1	Both 3 months	Newcomer to add functionality to the graph layout code.	Exposure to a new part of the system. Both in same onboarding session	
2	2 days	Newcomer to make changes to reports generated by the system.	Newcomer worked there for 3 months previously	
3	First day	Being primed for (as-yet) unspecified, maintenance requests.	Expert and newcomer had worked together before	
4	First day	Newcomer to make urgent changes to the UI component of the system.	Expert and newcomer had worked together before	
5	First day	Being primed for (as-yet) unspecified, maintenance requests to the back end.	Newcomer previously worked designing the original system	
6	First day	Newcomer to make small enhancements to the current network analyses.	Exposure to a new part of the system.	
7	7 months	General introduction to the system.	Supervisor in attendance	
8	First day	Newcomer to assess the codebase in terms of software quality.	Developer also in attendance	
9	7 months	Newcomer to make specific enhancements to the analysis functionality.	Exposure to a new part of the system.	
10	1 day	Newcomer to add a 'charting' functionality to the existing infrastructure.	Supervisor in attendance	
11	2 days	Newcomer familiarized with data-storage management application towards adding the charting functionality.	Newcomer used to work at the company. Two developers in attendance.	
12	7 months First day 3 weeks	One newcomer to make changes to correct a memory issue. The others were getting more familiarized to the system.	Newcomer used to work at the company. Two experts in attendance	
			Exposure to a new part of the system. All in same onboarding session.	
			Supervisor in attendance.	

system overall, not the specific part focused on in the sessions. Their overall levels of experience, and the similarity of the newcomers' and experts' responses, indicates that this group are indeed newcomers and not novices.

Table 2 shows that eight of the sessions occurred in the context of specific change tasks for the newcomer. Another session was contextualized by the newcomer assessing source code quality and the other three were more general introductions to the systems involved; not oriented towards a specific task. As befits a naturalistic field study of this type (Lethbridge et al. 2005), the pairing of newcomers and experts was left entirely to the organization involved.

4.4 Data Preparation

To prepare the data for analysis, and to develop familiarity with the data, all of the onboarding sessions and interviews were transcribed by the researcher as soon as possible after collection. For the audio-only interviews, this was straightforward, but the simultaneous videos of the onboarding sessions presented some problems. These were overcome somewhat by the use of the Matroska media file format (Matroska 2013) and VLC (VideoLAN 2013) to play the videos simultaneously in their own windows which could be resized as appropriate. This solution required insertion of timestamps into the transcription for synchronization purposes. Despite this ultimately satisfactory tool setup, the transcription was an inherently laborious process: The transcripts included details from the screen capture 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.

4.5 Tool Support for Analysis

The role of qualitative data analysis (QDA) software is to assist with organizing the study's collection of materials, concepts, categories and notes. From the various QDA packages on the market, ATLAS.ti v6 (Muhir 2013) was chosen for two main reasons:

- Its support for the analysis of video materials: Transcripts cannot capture every nuance of individual expression, so coding from the transcripts alone carried a risk of misinterpreting the participants' verbal contributions. ATLAS.ti supported the analysis by allowing the recordings and the transcripts to be viewed synchronously. This feature served to reduce the chance of misinterpretation (particularly with respect to 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, but was addressed by a technique called 'constant comparison of concepts' (see Section 4.6) which is supported in ATLAS.ti. Subsequent use of the constant comparison technique in this study was used to keep the breadth and depth of concepts in balance.

4.6 Analysing the Data

Early analysis in Grounded Theory (GT) takes the form of open coding. This is a process of asking questions of the data in order to break it apart into concepts, called codes. 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 visualizations (Riley 1996). Whenever a relevant incident or occurrence is found in the data, it is coded with a short descriptive label. Thus, the result of open coding is a set of concepts, or codes, that represent the body of data from which they were derived.

Each time an incident is coded, it is compared to every other incident with the same code. This technique is called constant comparison and is central to the GT analysis process. Constant comparison relies on having collected a varied set of data and allows any variation that exists within the data to be first discovered and then managed. The variations reveal the properties of the concepts and the dimensions of those properties (i.e. the range of ways each property can vary). The more variation there is in the data that has been collected, the more fully the properties and dimensions can be discovered. This process often leads the researcher to change the data collection protocol, in order to look for more, different cases for comparison.

However, it is important to have a balance between variety and saturation. In GT, saturation is a measure of the grounded-ness of the codes it produces. The more saturated the code, the more it is grounded in the data from which it was derived and the less it needs further data to support it. Indeed, the guidelines for Grounded Theory suggest that data gathering should cease when the resultant theory becomes saturated: that is, when further data fails to provide new codes and existing codes are already saturated. ATLAS.ti supports this process by hyperlinking each code to and from each piece of supporting data.

The coding process also involves comparing codes/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.

Constant comparison and memoing lead to the identification by the researcher of what are called axial codes, which label more abstract categories of concepts. These are the ‘bones’ of the theory that is being built and because they are directly linked to the codes from which they are derived, they help to make sense of the data and explain what is going on in the domain that is being investigated. GT also provides for a further stage of analysis, namely selective, or core coding, which is not always necessary. The core concept in this study is system comprehension.

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, which are notes that are again central to the GT process and are the building blocks of the emerging theory being built by the researcher.

4.7 Chain of Evidence

The following example shows a concrete instance of the process of coding, from a fragment of a transcript through to an axial code.

Figure 3a shows part of a transcript loaded into ATLAS.ti. During this session, the expert pointed out some unused code to the newcomer and gave the circumstances under which it might be reinstated. In the initial open coding stage, this was coded as “pointing out changing code”.

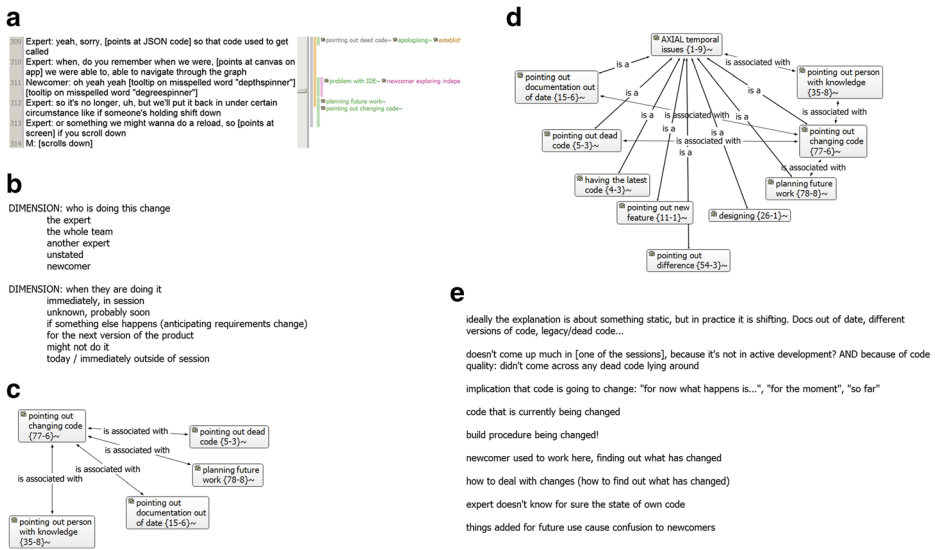


Fig. 3 **a** part of a transcript with codes attached, during the open coding phase. **b** part of the notes for a code, produced by the process of constant comparison. **c** representing the relationships between codes. The numbers associated with each concept indicate the number of examples of that concept in the data, followed by the number of relationships to other concepts. **d** the relationships between the temporal-issues axial code found in the study and nine other concepts, also found in this study. Many different types of relationships are possible in GT and are supported by ATLAS.ti, including association, is-a and contradicts relationships. **e** memoing for the “temporal issues” axial code, capturing details that became important to the grounded theory

This open code appeared in several transcripts. When an existing code was applied to a new incident, the new incident was compared to the previous incidents to look for variations between them. Figure 3b shows part of the notes for the “pointing out changing code” code in ATLAS.ti. Specifically it shows that these incidents varied in terms of who would be changing the code and when the change was planned, among others. These became the dimensions of the code. When new incidents no longer reveal new dimensions, the code is said to be saturated.

The transcript itself shows that “pointing out changing code” co-occurs with “planning future work” and was preceded by “pointing out dead code”. This is also captured in ATLAS.ti by noting relationships between codes, which can be visualized as a network (Fig. 3c).

Over time, clusters of related codes emerge from the network. This leads to the generation of a more abstract, axial code that captures a higher level name for the cluster of codes. Axial codes then form the basis for the grounded theory.

Axial codes also have memos. Figure 3e shows part of the memo for the “temporal issues” axial code, leading to the insight that ‘the constantly changing nature of the codebase under explanation was an important aspect of onboarding’.

5 Results

The findings are presented in terms of the four different viewpoints/perspectives which the experts employed in order to represent the code under discussion to the newcomers during the sessions. These are discussed, expanded-upon and illustrated with representative quotations from the session participants in the following subsections. Several of the findings presented in

this section may be considered provisional because they were more indirect interpretations of events (not directly observable events). Consequently, we present the results of a follow-up questionnaire study undertaken to assess the ‘fit’ of these more provisional findings in Section 5.5, as recommended in Corbin and Strauss (2008).

The four viewpoints of the code, and the activities that contributed to them, are illustrated in Fig. 4. They are:

1. The Structural view: These refer to “What is there”, describing and characterizing the high-to-mid level components of the system.
2. The Algorithmic view: These refer to “How it works”, describing the inner workings of the code towards achieving some functionality, specified by the expert.

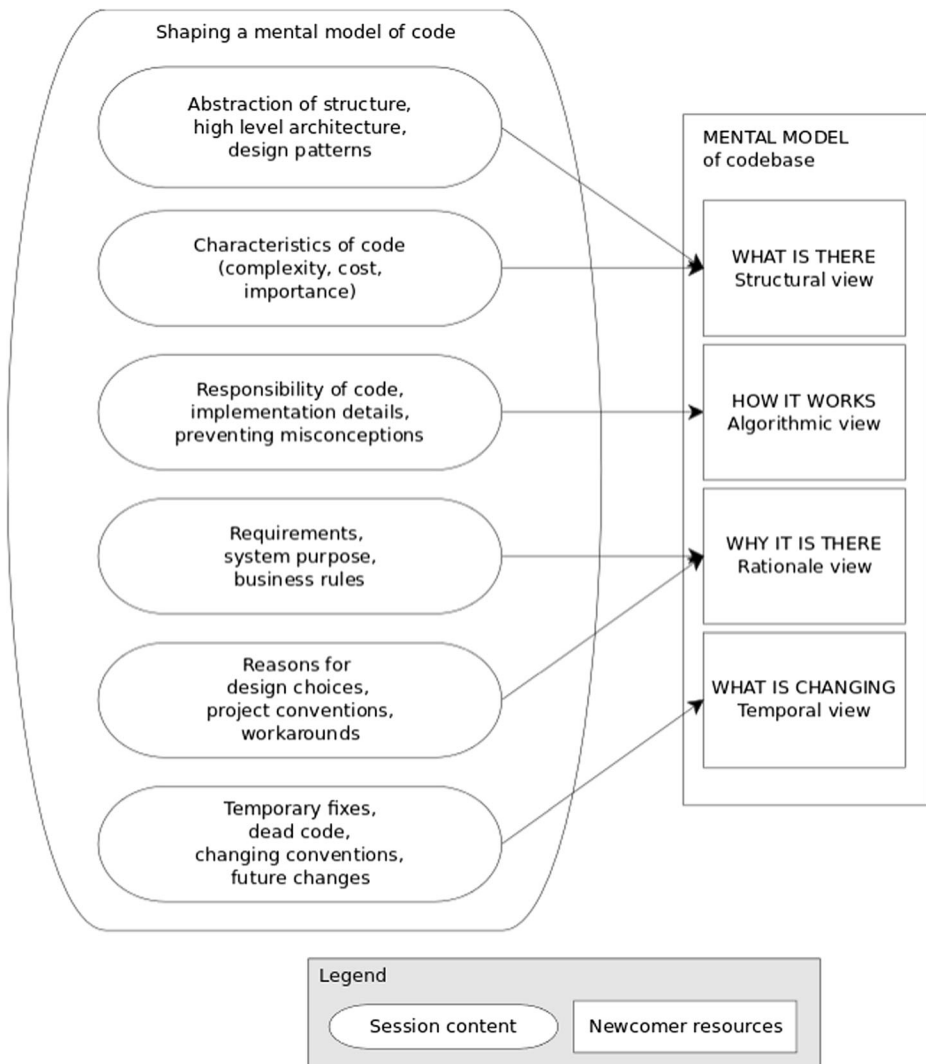


Fig. 4 An overview of the findings

3. The Rationale view: These refer to “Why it is there”, describing the system requirements and the design principles that led to the current design.
4. The Temporal view: These refer to “What is changing, what has changed and what will change” in the system and how it impacts on the code.

In the following section, each of these views is expanded upon. In each case quotations are presented to illustrate the observations and to provide traceability from data to results. These quotations are followed by the name of their associated codes and the number of times that code was noted in the data-set, giving an indication as to the prevalence of each code. The ordering of the groups presented here does not imply any corresponding order in the sessions; in practice, explanation from all four viewpoints was intertwined within the sessions, with the exception that many sessions began with a high-level structural view of the systems’ architectures.

5.1 The Structural View

The Structural view provides a high-level view of the components of the system, encompassing abstraction and characterization of the code. It is typically more prevalent at the start of the onboarding session, but is used throughout the session.

5.1.1 Abstraction of Structure

These abstract views of the code range from high-level identification/descriptions of the major (possibly distributed) components of the system, their relationships, and the system’s architectural patterns to finer granularity identification/descriptions of design patterns. An example of a high level description at the C# project level is:

S3 expert: *So if you collapse down all the (C#) projects, I'll give you an overview of all the projects and how they work together... [The S3 newcomer then collapses all projects]... [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... ("high level explanation", 54)*

At lower levels of granularity, the expert may explain code in terms of design patterns, which, with one name, expresses the rationale for, existence of, and relationships between, a group of classes (Gamma et al. 1995). The data obtained in this study confirms, as would be expected, that several design patterns, such as Observer (or Listener), Command and Visitor, have become industrial standards, and they occur prevalently over the data set. At other times though, the names of design patterns are not referred to directly, but are alluded to more by intent, suggesting that this traction is not always there.

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

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... ("using design patterns", 46)*

At each level, this provides the newcomer with a rough outline of the ‘shape’ of the code-base, and a way to group and refer to related code, reducing their need to examine all the details. The value of this was evidenced by the subsequent interviews held with the newcomers:

S6 newcomer: *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.* ("ability to find code", 29)

5.1.2 Characteristics of Code

In addition to identification of the macro-components of the code, experts provide information about the characteristics of various pieces of the code-base. These can include its complexity, its cost, and its relative importance, in overall terms or within the current context. Some characteristics, such as performance, are more factual, while others, like its relative importance are more subjectively stated.

With respect to complexity, both simple and complex code is pointed out, but experts point out simplicity more often than 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.

S1 expert: *[This class is] just really simple, there's nothing to it, it's an ID and a hashtable.* ("pointing out simplicity", 91)

S3 expert: *[This] is the core logic for mail-merging all the different types of documents, it's really complex, really messy, but it does the job.* ("pointing out complexity", 37)

The expert often 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 can include processing power and execution speed but can also encompass other, more diverse, measures of cost like development time. Cost are considered in absolute or relative terms by the expert.

S6 expert: *[Recursive method] goDeeper is where we go (waves hands) a level out, so goDeeper is the expensive branch.* ("pointing out cost", 24)

S5 expert: *Pulling down that tiny table every ten seconds is nothing.* ("pointing out cost", 24)

By pointing out a snippet of code, the expert implicitly communicates its importance, but often this is emphasized explicitly, using visceral terms like ‘meat’, ‘guts’ or ‘heart’. Code is considered important if it implements a critical function of the system.

S3 expert: *...but just take note, actioner host is a critical page.* ("pointing out importance", 55)

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...* ("pointing out importance", 55)

Importance may also be ranked relatively, where some code is stated as more important than other code:

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.* ("pointing out importance", 55)

Likewise, experts also point to areas of code that can be considered less important. Code seems to be considered less important to the session if it implements internal functions not contributing to the system's core purpose (such as logging or validation) or is an example of repetitive 'template' or 'boilerplate' code.

S11 expert: *...just all general initialisation and setup stuff. ("pointing out importance",*

S3 expert: *(selects SQL code) This is just boilerplate code for creating a table. ("pointing out importance", 55)*

Taken overall, the Structural view provides 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 of four views that experts present during onboarding.

5.2 The Algorithmic View

The Algorithmic view focuses on how features are implemented. Here experts describe or demonstrate a selected code responsibility, and connect it with its typical execution, in varying levels of detail.

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... (some time passes and the S3 newcomer opens MakePayment.aspx)... That's the page you saw for entering the credit card information... ("demonstrating software", 80)*

Having identified the responsibility of the code, the expert goes into more detail about the inner workings of the implementation. This is typically broken into steps, with the amount of detail given for each step varying: sometimes a core line, variable or object will be described in careful detail, but other code, comprising many lines, can be grouped and summarized in one phrase.

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..("summarizing code", 158)*

Newcomers report that it was not generally useful for the expert to go through every line of code at this level; they only deemed it useful for unusually complex code. In congruence with these newcomers' needs, experts do not typically describe the workings of all the code. However, instead of focusing on complex code, they tend to focus on either the 'core' execution flow through the system, parts of the code that they were recently involved with, or code relevant to the newcomer's first task.

There are three interesting characteristics with respect to the content of the Algorithmic View. Firstly, newcomers are able to infer some of this information from the code itself, so this aspect of the discussion tends to be interactive, with the newcomer checking their assumptions against the expert's knowledge. Examples of this include:

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. ("making observation", 132)*

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. ("making observation", 132)*

Secondly, 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.

Finally, experts often tried to prevent misconceptions on the part of the newcomer during their discussions of the Algorithmic View, by highlighting plausible but incorrect models of the system's execution.

S7 expert: *I'm just connecting different parts and it doesn't add another edge, it just changes the existing one. ("preventing misconception", 91)*

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. ("preventing misconception", 91)*

5.3 The Rationale View

The Rationale view covers the system's requirements and the reasons that led to the system's current design. The purpose of a system and its design decisions may be highly obscure (depending on the domain and documentation available) and here the experts try to address these difficulties for the newcomers.

5.3.1 System Requirements

The most common way in which the experts illustrate the system's requirements is by walking through the executing system with the newcomer, as per the first example given in the Algorithmic View Section (5.2). However, it should be noted that the requirements provided here are more encompassing in nature, referring to the functionality of large parts of the system. Thus, they differ from the selected code responsibilities/functionalities that were mapped to how-to logic in the Algorithmic View. The newcomers seemed to appreciate this perspective:

S8 newcomer (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. ("giving purpose of system", 34)*

On other occasions the experts stated the system requirements outright or by means of diagrams or sketches that focused on the business rules and processes implemented by the system. Again, this approach seemed to be appreciated by the newcomer.

S2 expert (*sketching diagram of stock transit*): So you dispatch 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. ("explaining business rules", 42)

Overall, requirements and context are largely passed on by verbal explanation in these sessions. Other than the occasional informal diagram, not a single session made use of formal requirements documentation, and no newcomer requested this form of documentation.

5.3.2 Design Rationale

Design choices are in evidence throughout systems, from the highest level architecture down to the use of variables. The highest level structuring of the code is often presented in terms of an architectural pattern, as described in Section 5.1.1 and when so, it is generally assumed that the newcomer is familiar with the generic rationale for choosing such a pattern without verbalizing it. Other, lower-level design principles, which are less clearly named, result from the requirements and domain, and are more likely to be verbalized by the expert.

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. ("giving design philosophy", 48)

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. ("pointing out project conventions", 66)

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]. ("giving a reason", 170)

A common trend in the onboarding sessions was the verbalization of non-obvious design choices, which often had their roots in problem workarounds or unexpected business rules. Without knowing these reasons, the newcomer may be tempted to 'correct' code in ways that would lead to unproductive regressions.

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. ("giving a reason", 170)

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. ("explaining business rules", 42)

S6 newcomer (*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. ("newcomer confusion", 80)

Another category of design choices are changes that occurred in the design over time, leading to non-obvious (current) design choices: The reasons for the original choice no longer stand, but it would be too difficult (or not worthwhile) to change the code. Understanding this, the newcomer can avoid wasting time searching for currently valid reasons for the design, or using the previous design as a template for future work. This category of information can be classified as a Rationale View but could equally be categorized as a Temporal View (see Section 5.4).

S3 expert: *[One project] started in use as web applications in there but now we put them into C development.*

S3 newcomer: *(opens project) 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. ("asking question", 258, "understanding why", 22)*

Overall, this set of quotations illustrates the newcomer's interest in design choices that do not immediately make sense. This was a prevalent theme during the sessions, again suggesting that it was of focal interest to the newcomers.

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. ("asking question", 258, "understanding why", 22)*

S3 newcomer, *(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. ("understanding why", 22)*

5.4 The Temporal View

The majority of software is under active development or maintenance. As a result, the codebase that the newcomer must learn is not static, but includes temporary fixes, dead code, changes to code conventions over time and changes planned for the future. The Temporal view informs the newcomer about previous, current and future development work that affects the code, providing a crucial extra dimension to what would otherwise be a static snapshot of a codebase. As active developers, experts are typically aware of these longer-term ongoing changes to the code-base and can express them.

Informing newcomers of these activities helps them to make sense of the changing system landscape. One category of ongoing change is the temporary fix. Such fixes are expected to change again in the near future and, being temporary workarounds, are poor examples for the newcomer to follow. The expert may justify poor practice with a reason for the necessity of the temporary fix.

S2 expert: *I'm just hard-coding a run at the moment because they didn't have it at the time.* ("pointing out changing code", 77)

The code-base 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 the presence of the dead code, giving some insight into the team's practices. Encountering dead code may also lead the expert to plan future changes to remove it (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... 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.* ("pointing out dead code", 5, "planning future work", 78, "pointing out changing code", 77)

Importantly, project-conventions may have changed over time, resulting in two or more conventions co-existing in the current code-base. 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 are current and should 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.* ("pointing out changing code", 77, "pointing out project conventions", 66)

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.* ("pointing out difference", 54)

Experts are also aware of changes planned into the future, 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 through in-the-next-version to not specified at all.

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.* ("ignore this", 16, "pointing out changing code", 77)

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.* ("planning future work", 78, "pointing out changing code", 77)

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; these codebases were relatively static during the sessions and planning future changes was the main source of temporal content.

5.5 Result Refinement: Evaluation of Fit

In order to assess the fit of the data to findings which relied on a more indirect interpretation of the data (Corbin and Strauss 2008), participants were invited to complete another questionnaire. Specifically, the role of that questionnaire was to ensure that findings based on such interpretation were not overstated or based on misunderstandings. Fourteen of the original 23 participants agreed to fill in the questionnaire: six participants that played the role of experts, six that played the role of newcomers, one who had both roles and one who was an observer in a session. The profiling question at the start of this fit questionnaire showed that the majority of respondents had been part of an onboarding session within the last month.

The questionnaire presented the participants with a list of statements (see Table 3) and they were asked to agree or disagree, using a five-point Likert scale (strongly agree, agree, neither disagree or agree, disagree, strongly disagree). The resultant data is also presented in Table 3.

As can be seen from the table, the questionnaire results suggest that its useful if the expert names chunks of code and explicitly addresses plausible misconceptions. Indeed, results from the questionnaire support most of the findings from the main study in general, with the exceptions of questions that assessed whether the expert should point out dead code, go through all lines of code and all lines of complex code. Six of the participants thought that experts should go through all lines of code whereas only two thought they should not. This conflicted with the findings from the main study where newcomers thought that this was not useful and with the intuition that most systems are too large to go through line-by-line. In terms of going through complex code line-by-line, only seven respondents to this questionnaire thought it useful with two disagreeing and five neutral. Again, newcomers in the main study thought this was a worthwhile practice. Finally, as regards ‘pointing out dead code’, the questionnaire did not support this as a useful practice. Even though the main study did suggest this as useful, it is important to note that this was observed only 5 times over the 12 onboarding sessions, suggesting that it is not a particularly prevalent onboarding activity.

6 Discussion

This discussion section first interprets the results of this study for practitioners, suggesting how they might leverage them to inform their onboarding practice. These interpretations are considered in the light of the current literature; an important consideration given that the Corbin and Strauss variant of Grounded Theory adopted derives its results primarily from the data with only general guidance from literature. The section then proceeds to discuss the implications of the results for researchers, concluding with the threats to validity encountered in the study.

6.1 Guidance for Onboarders

The Importance of Rationalizing Design Decisions In the sessions reported on here, experts proactively provided a rationale-based view of code, including both function-based rationales and design-decision rationales (see Sections 5.1.1 and 5.3.2, and Table 3). These seemed to be valued by the newcomers, with several of the newcomers probing design decisions, particularly with respect to obscure design decisions. Thus, it seems, both experts and newcomers agreed on the value of this kind of information, suggesting that it is a valuable component of onboarding.

Table 3 The fit questions for findings that were more-provisionally established in the onboarding sessions

Statement	Expected Response	Actual Responses	Supported
Structural Views			
It is useful to the newcomer if the expert describes the high-level Architecture	Agreed	Strongly agree: 9 Agree: 5	Yes
It is useful to the newcomer if the expert points out the design patterns in use	Agreed	Strongly agree: 4 Agree: 7 Neither: 3	Yes
It is useful to the newcomer if the expert names chunks of code	Agreed	Strongly agree: 3 Agree: 10 Neither: 1	Yes
It is useful to the newcomer if the expert points out simple and complex areas of code	Agreed	Strongly agree: 3 Agree: 8 Neither: 2 Disagree: 1	Yes
It is useful to the newcomer if the expert points out costs (speed, memory)	Agreed	Strongly agree: 2 Agree: 6 Neither: 5 Disagree: 1	Yes
If the expert informs the newcomer which areas of code are off-limits, the newcomer will find this useful	Agreed	Strongly agree: 2 Agree: 7 Neither: 3 Strongly disagree: 2	Yes
Algorithmic			
It is useful to the newcomer if the expert goes through complex lines of code line by line	Agreed	Strongly agree: 1 Agree: 6 Neither: 5 Disagree: 2	No
It is useful to the newcomer if the expert goes through all the code, line by line	Disagree	Strongly disagree: 1 Disagree: 4 Neither: 7 Agree: 2	No
If the expert points out possible misconceptions, the newcomer will find this useful	Agreed	Strongly agree: 4 Agree: 9 Disagree: 1	Yes
Temporal			
If the expert points out temporary fixes in the code the newcomer will find this useful	Agreed	Strongly agree: 2 Agree: 8 Neither: 3 Disagree: 1	Yes
If the expert points out dead code, the newcomer will find this useful	Agreed	Strongly agree: 4 Agree: 4 Neither: 4 Disagree: 2	No
Project conventions may change over time, resulting in two or more conventions co-existing in the codebase. If the expert discusses this, the newcomer will find this useful	Agreed	Strongly agree: 4 Agree: 7 Neither: 3	Yes
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	Agreed	Strongly agree: 3 Agree: 7 Neither: 3 No answer: 1	Yes
If the expert describes the longer term on-going changes to the codebase the newcomer will find this useful	Agreed	Strongly agree: 1 Agree: 10 Neither: 2 Disagree: 1	Yes

Mapping code to domain functionality is a long-standing tenet of program comprehension (Shneiderman and Mayer 1979; Detienne 2002; Pennington 1987) and feature location (Dit et al. 2011; Rubin and Chechik 2013). Likewise, design rationale has long since been considered integral to de-novo software development (Gorton 2006; Smith-Atakan 2006).

However, the explicit empirical evidence for the importance of obtaining design decision rationales in program comprehension literature is surprisingly mixed: Ko et al. (2007) observed that developers sought rationale information from knowledgeable co-located co-workers, while Sharif et al. (2015) found that open-source developers often asked for rationale information through mailing lists (Why questions on System-Design topics). But von Mayrhauser et al. (1997) found that developers voiced very few why hypotheses concerning the purpose of a design choice. Likewise, Ratanotayanon and Sim (2006) found newcomers asking very few, if any, why questions of the experts in their studies.

A possible explanation for this inconsistency is that developers' questions/hypotheses are matched to the perceived capabilities of their information sources. Rationale questions can often only be answered by a true system expert and von Mayrhauser et al.'s empirical study looked at developers interacting with source code only. Likewise, Ratanotayanon's participants did not have access to true system experts either. The results obtained here align more with the importance of design-decision rationale in Ko et al. (2007) and Sharif et al.'s (2015) work, illustrating that it is a sought-after element of system understanding when a source of such information (an expert) is available.

The Importance of Temporal Knowledge The majority of current software systems are under active development or maintenance at the time of onboarding and will likely be further developed and maintained into the future. The results presented in Section 5.4 and Table 3 suggest that experts in the empirical study focused on this non-static aspect of the system in their descriptions, noting past changes and practices, the permanent and temporary changes currently being undertaken, and changes planned into the future (although future plans were less obvious in sessions 2 and 3, where the code base was not under active development).

Intuitively this seems like an important perspective because the source code is often cited as the definitive source of comprehension information (Sillito et al. 2008; Ko et al. 2006), but that definitive source is constantly shifting over time. This adds an orthogonal dimension to the comprehension process, as developers must now remember which parts are out of date, unused, currently in flux, likely to change and why. Obtaining this information (from the expert) is the first step in forming that knowledge base, although it is interesting that no newcomer expressly noted the utility of this information.

The majority of empirical studies in this area have required comprehension or modification of a static code artefact as a once-off (Pennington 1987; Soloway and Ehrlich 1984; Shaft and Vessey 1995) or concerned one programmer understanding a system in isolation (von Mayrhauser and Vans 1993, 1995a, b). In these scenarios, there is no motivation for the programmer (newcomer) to consider temporal aspects, no source of temporal information beyond the current code itself and, where the code was written specially for an experiment, there are no artefacts of temporal activity to trigger Temporal view considerations. A notable exception is the work by Ragavan et al. (2016, 2017), where they assessed and modelled programmers undertaking maintenance tasks that leveraged a repository of previous versions of the system (referred to as 'variants' in their work). In their 2016 study, novice programmers were presented with a maintenance task that was phrased in a manner suggesting it involved reverting back to a previous version of the code, in part. Hence the participants searched through the variants for direction, and their navigation strategies were uncovered. In their 2017 paper Ragavan et al. developed data/algorithmic models that were congruent with the variant-navigation task, towards better tool support. Another work that focused on temporal views of code was Murray and Lethbridge's (2005a) study of temporal 'snapshots'. That work focused

on experts' descriptions of their systems (Murray 2006) where moment-in-time views of the system (snapshots) were extended into 'Temporal Detail' patterns, in which historical (conceptually whole) moments in the evolution of the system are captured.

However, this latter work only refers to key historical moments suggesting a focus on major changes only and does not refer to on-going or envisaged future changes, as identified here. The variation perspective proposed by Ragavan et al. is more relevant in that it seems to be of finer granularity but, even then, it does not explicitly address future planned changes, or the rationale commentary that an expert guide can provide: rationales that explain why certain code is no longer used, and the trajectory of changes over multiple versions (temporary, moving from one convention to another).

Focusing on Walkthroughs of Complex Code Chunking into larger macro-level abstractions like plans (Brooks 1983; Pennington 1987; Boehm-Davis et al. 1996) is a de-facto component of the program comprehension literature, and was reflected in this study: Experts frequently walked through code algorithmically to explain its workings and intent. However, this focus was often on core code and code that the expert had visited recently, the latter suggesting that the experts directed attention to what they understood, as opposed to what was of value to the newcomer (see Section 5.2). In contrast, newcomers seemed happy to walk through the code themselves and thought that walkthrough effort should instead be reserved for complex lines of code (see Section 5.2 and Table 3). This suggests a communication mismatch between expert and newcomer that should be avoided in onboarding.

This aligns with information foraging theory (Lawrance et al. 2010) where newcomers, being experienced coders, felt they could work out the details of most of the code for themselves and wanted to leverage the expert for more difficult information. That is: where the opportunity (the expert) is available, more valuable prey (complex code) should be seized on. Indeed, this is also reflected in our findings regarding the "valuable prey" of design rationale. For less complex code, newcomers are aware that it will be available to them outside of the sessions and seem confident of their ability to interrogate it independently, as evidenced by the interactive nature of their discussions with experts when walking through that type of code during the onboarding sessions.

Describing Potential Misinterpretations Experts described plausible misinterpretations that could be drawn from the code base. These include conventions that were once applied, and are still persistent in parts of the codebase, but are not to be applied going forward (see Section 5.4). They also include subtleties in algorithms that lead to ambiguity and plausible, but incorrect, rationales for design decisions (see Section 5.2). Our literature review offered up no literature on the potential value of disambiguating potential misinterpretations during code comprehension, probably due to the pull nature of the existing literature where misinterpretations would be simply categorized as programmers' mistakes.

Adopting a Top-Down Approach Unsurprisingly, experts provided a top-down view of code, starting with the high-level structure of the codebase (see Sections 3.1 and 5.1.1) before going into more detail. Newcomers reported that the high-level view was valuable session content, as it would allow them to navigate the codebase with greater ease subsequently. Low-level information was perceived as of lesser value because newcomers thought they could reconstruct most of it themselves when required.

In the literature, a top-down comprehension approach is seen as more optimal, with the alternative bottom-up approach being denigrated by Brooks (1983) as ‘degenerate’. O’Brien et al. (2004) and von Mayrhauser et al. (1997) linked a programmer’s ability to work top-down with their prior knowledge of the domain, knowledge that the experts in these sessions presumably had. Transfer of this knowledge allowed newcomers to adopt a more as-needed strategy (Littman et al. 1986), effectively navigating around the code to the pieces they needed for their task context. The alternative is ineffective, opportunist navigation, associated with unsuccessful maintenance, or having to develop a wider, more systematic understanding of the system (Robillard et al. 2004), associated with more successful maintenance, but greater effort.

Another notable aspect of the results is the reluctance of the onboarders to use the various resources available to them. For example, in walking through the code, a debugger would seem to provide appropriate support. Likewise, various documentation and project design diagrams might help in the discussion of design patterns and architectural patterns. However, debuggers and project documents were not leveraged in these onboarding sessions (see Section 5.2). While diagrams were sometimes employed, they were created during the sessions from scratch, and were highly informal representations. Again, in line with information foraging theory, this suggests that the perceived cost of running the debugger, or finding the documentation, is seen as disproportionate to their added value in onboarding, where the expert does not need to check the information but is simply presenting already acquired knowledge.

6.2 Implications for Researchers

Most fundamentally, the results presented in this paper suggest a refinement of the existing theories of information seeking and program comprehension. In terms of design rationale, the findings presented here reinforce the impression that design rationale is important in the context of program comprehension, in line with the findings of Ko et al. (2007) and Sharif et al. (2015). In addition, they characterize the types of design rationale information exchanged: non-obvious work-arounds, team conventions, obsolete design choices and a focus on design choices where the rationale is open to misinterpretation.

In terms of the Temporal view, this work also extends existing findings. Unlike earlier work (Murray 2006; Ragavan et al. 2016) the temporal perspective presented here considers current change as fluid, co-existence (for example, of conventions) during current changes, planned future change, temporary change, and the explicit direction of change over multiple versions.

These extensions raise important questions for researchers. Design rationale seems important for both experts and newcomers, and as such, it seems like a core program-comprehension perspective in practice. However, only the experts focused on temporal knowledge: the newcomers did not note it as either helpful or unhelpful. Hence its value to the newcomers remains unclear. This is also the case regarding the ‘potential-misinterpretation’ information that the experts emphasized in their descriptions of their systems: again, the newcomers did not note this as either helpful or unhelpful.

This suggests that researchers carry out further studies to ascertain the actual value of these two perspectives to newcomers and do so at greater levels of granularity. It may be, for example, that out-dated, but persistent, conventions that lead to inconsistencies in the system going forward are important to know about, but that algorithmic misinterpretations are less important, as the newcomers are capable of walking through the code to resolve this latter

category of ambiguity by themselves. Additionally, if the value of various temporal and misinterpretation-avoiding information is validated, then additional research challenges arise, such as how to visualize this information for developers.

In emphasizing design rationale, the findings incorporate design patterns (Gamma et al. 1995) and architectural styles (Shaw and Garlan 1996), as relevant system chunks during newcomer ramp up. Thus, it seems that the Design Pattern/Architectural Style community's efforts towards a common vocabulary of larger scale abstractions, addressing non-functional requirements, are gaining momentum. But the findings from this study suggest that the languages they seek to create are, as yet, incomplete, or at least not widespread in their adoption (Bass 2007; Clements et al. 2002): experts often found it difficult to name the abstractions they were endeavouring to explain. This suggests that the research community redouble their efforts to identify and standardize the common vocabulary in these communities.

Interestingly, task-context has been shown to have an impact on the results of other program comprehension studies: Programmers tasked with familiarizing themselves with code obtained a structural perspective on the code-base, while programmers tasked with making a change to the code obtained a more functional perspective (Pennington 1987). Here four of the sessions (S3, S5, S7 and S8) could be characterized as familiarization, and agnostic to software change. The other eight were performed in the context of one or more imminent code change(s). However, the trends in the data were similar across both these sets of sessions suggesting that, in onboarding at least, task context does not impact on the views emphasized. It should be noted though, that this is a provisional result, based as it is on two small sets of sessions and should be buttressed by further empirical studies.

6.3 Threats to Validity/Reliability

The onboarding sessions themselves all happened in a naturalistic fashion, in the onboarders' workspace, and were directed at the onboarders' system(s). In addition, both experts and newcomers were experienced professionals and these factors raise the credibility of the results (Lincoln and Guba 1985). But only 12 experts and 15 newcomers were studied, over 12 in-situ onboarding sessions. This is a small data-set, necessitated by the limited availability of professional developers to participate in empirical studies and by the timing requirement: participants had to be about to carry out onboarding of a new employee. As such, the scale raises questions about the transferability of the results. However, it should be noted that these joint requirements demanded capturing data in Canada, the US, England and Ireland over an 18-month period.

Another issue is the size of the systems under study during these onboarding sessions. Von Mayrhauser and Vans (1995a), would have referred to such systems as medium sized, but in the last 23 years, the scale of software systems has grown substantially. We could not control the systems in these in-situ studies and they could legitimately be referred to as small systems in today's terms. This is particularly true of those at the lower end of the scale and it is likely that those systems could be studied systematically, rather than in an as-needed fashion, as demanded by larger systems (Littman et al. 1986). Studies on larger systems need to be performed to buttress the representativeness of the findings.

Participants were also recruited based on their willingness to participate. This raises the possibility that those willing to participate had more confidence in their onboarding session techniques than the general population of developers. This represents a threat to the

transferability of the findings (Lincoln and Guba 1985), but the alternatives - having developers participate against their will or without knowledge of recording - are clearly unethical. Additionally, a wide variation in session styles and expert preparation suggests that the recordings captured a representative, if not an encompassing, sample of behaviour.

The degree to which the onboarding sessions studied can be characterized as information-push, as opposed to information-pull, is open to debate. Hence, for each of the sessions, we performed a post-hoc analysis of the number and the type of questions that newcomers asked the expert during onboarding. The results are summarized in Table 4.

The table suggests that the sessions were quite varied in terms of interactivity: Indeed, in some of the sessions, questions were few-and-far between, whereas in others, newcomer questions were very frequent, echoing Begel and Simons findings on the importance of communication skills for *novice* developers in software development (Begel and Simon 2008a, b). In order to clarify the degree to which the sessions were information push/information pull, with respect to software comprehension, the questions were studied qualitatively, in greater depth. The vast majority of questions were identified as falling into two categories:

- Environmental set-up where the newcomer was asking for help concerning the configuration of their machines, in terms of software development tools like version control repositories, and IDEs;
- Rhetorical questions, where the newcomers answered their own questions themselves, based on information already given them by the experts. They often allowed the expert a chance to correct them if they were wrong.

Questions of the former type are information-pull in orientation, but they do not directly refer to understanding the system. Instead they refer more to understanding the organization's system-development environment. The rhetorical questions are more nuanced: they could be considered information pull, in that they pull confirmations from the expert. But, as they all referred to information previously pushed to the newcomer by the expert, they could probably be considered confirmation mechanisms, more reflective of the clarity of the information pushed, and certainly targeted at the information pushed.

Interestingly the context of this study by-passed one of the main problems reported when software developers try to leverage colleagues as an information source: that of identifying knowledgeable colleagues (Mockus and Herbsleb 2002). Here the knowledgeable colleague is assigned in advance. Indeed, although not included in the findings in this paper, the experts in this study sometimes identified other point-of-reference experts for different parts of the system: what Ko et al. (2007) refer to as the social knowledge underpinning a system. These observations do not overcome another problem noted (McDonald and Ackerman 1998): that identified colleagues may have poor explanation skills. In fact, the work reported on here can

Table 4 Number of questions asked by the newcomers during their session

	Questions	Session Duration	Questions/min	Questions/h
Min	1	18	0.06	3.33
Max	44	96	0.82	49.33
Mean	25	51	0.48	28.55
Std. Dev.	13	21	0.21	12.42

be seen as trying to partially address this problem by providing insight on the content that should be focused on during onboarding sessions.

A threat to the credibility of the study is that five of the sessions were set in academic research contexts. It is entirely possible that experts, who were also academics, would have different insights into the information that should be communicated and/or would leverage their pedagogical expertise to direct the session. However, in all five cases, the academics involved were full-time researchers, in post-doctoral positions. They were not lecturers and their research role focused, to a large degree, on the system being studied. In addition, three of these academic sessions were led by an expert who had 2.5 years' experience in industry (albeit developing a separate software system).

Additionally, the presence of the observer and recording equipment during the onboarding sessions might have led to researcher presence-effects (Easterbrook et al. 2008), where participants alter their behaviour based on the presence of the researcher (Adair 1984). Indeed, there were some indications that participants occasionally became explicitly aware of the recording equipment. This tended to happen after a joke or a mistake. However, such events were infrequent in the sessions overall, and diminished over the course of the session, suggesting that participants became more immersed in the session and less aware of the observation as time went on.

Further threats to validity arise in the questionnaire section of the study. Firstly, there was a considerable delay (ranging from two to four months) between the on-boarding sessions and the administration of the 'fit' questionnaires. This delay could have impacted the respondents' recall of their sessions (Ericsson and Simon 1980). However, this delay was planned to allow respondents time to modify the code under discussion in the session, thus enabling them to better evaluate the information presented in the sessions. In addition, a number of the respondents had been involved in further onboarding sessions since the observed session, suggesting that their responses may have reflected wider onboarding experience. A related issue is that of self-reporting, also associated with questionnaires. Self-reporting is regarded as open to bias, forgetfulness, and trying to make the respondent look good to the researcher (Northrup 1997). In this study, the researcher did not have an initial position and so the potential to satisfy the researcher was limited. In addition, some of the questions posed in the fit questionnaire were deliberately expressed as the inverse of the hypotheses being tested, to address this issue.

These validity issues suggest that work should be undertaken to test these findings on a larger population, as future work. Such a study could expand the fit questionnaire derived here to assess all the findings and this could be used as a basis for a survey of industry professionals. This survey would have to be prefaced by additional open questions on the important information to communicate during onboarding, seeking any additional information types, not discovered by this study, that might be identified. However, this would not address the self-reporting and delay issues associated with the questionnaire. Complementary studies are thus important, where questionnaires are presented immediately after onboarding sessions or in-depth interviews are used to investigate the essential information to communicate to newcomers.

Finally, all the coding and categorization of the data was performed by the first author, raising issues regarding the reliability of the interpretation. However, this analysis was subject to regular (monthly) reviews by the second and third authors, where both played the role of devil's advocate. That is, for any interpretation that did not seem apparent, the first author had to make the data and rationale for that interpretation explicit. The second and third authors reviewed these and often suggested alterations and/or refinements to the coding/categorization.

Additionally, where there was any doubt as to the interpretation of codes in the dataset, the fit evaluation was employed to refine the findings.

7 Conclusions and Future Work

This research originated from two related issues: firstly, that understanding unfamiliar software is a difficult task for even experienced developers and, secondly, that the program comprehension literature has focused nearly exclusively on the information-pull perspective. In that perspective, developers are trying to pull the information they need from system artefacts such as source code. In the limited number of studies of onboarding in the literature, little attention has been paid to the content of onboarding sessions, thus limiting the insights that have been found on the types and utility of the information that experts push to newcomers. However, the content of in-situ onboarding sessions merits study because these are not only an important part of real-world program comprehension processes, but they also provide a relatively natural opportunity to investigate the information actively pushed by expert developers towards newcomers.

While the findings from any such field studies must be considered provisional, due to the limited number of cases studied, the presence of recording equipment and the delay/self-reporting issues associated with the questionnaire, they do complement and augment findings from the more-studied, information-pull perspective. In terms of structural information, both perspectives suggest the importance of abstractions although this study suggests that abstractions addressing non-functional requirements play a larger role than previously reported. Likewise, both perspectives suggest the importance of an algorithmic view, but this study suggests more nuance: that newcomers appreciate the Algorithmic view only for complex code whereas experts tend to focus on code they have recently worked on, the core execution path through the system or code relevant to the newcomers' upcoming tasks.

In the existing literature, finding the functionality associated with code is a core part of program comprehension theories but the need for design rationale information during program comprehension has had a more limited evidential basis. Similarly, this work extends our current knowledge of the temporal information imparted by experts. Here, the shifting nature of the system over time is prominent in experts' explanations, whereby past, current and future changes are discussed in order to allow newcomers to appreciate time-dependent subtleties of the system, such as changes in conventions over time and temporary fixes that should not be relied on. The closest work in the literature to this work is that of Ragavan et al. (2016), where the utility of past temporal information is probed for a re-use based, information-pull, maintenance task.

Finally, this study supports the continued importance of plans and beacons for comprehending and discussing code. However, languages and programming paradigms have developed since beacons were first investigated, and systems have grown larger. These factors are evidenced by the chunking of plans at package and project level found in this study.

Future work will be directed at growing the body of empirical evidence in this area, perhaps by gathering more data from onboarding sessions or from online videos of experts describing their systems (MacLeod et al. 2015). It will assess the factors that prompt certain views: for example, there is some evidence from the study suggesting that the importance of the

Temporal view is related to team-size and ownership policies. Likewise, as researchers find and validate more accurate assessment instruments for developer experience, it would be interesting to look, in a more finely grained way (Feigenspan et al. 2012), at how past experience/ability affects the communication between the newcomer and the expert. It would also be interesting to study delocalised onboarding sessions in which communication is electronically mediated as standard. This would lessen any observer effects but would bring to the fore other issues with offshore development projects such as cultural differences and language barriers.

Finally, while some experts prepared for sessions by previewing the relevant code, others did not, and it was common for them to mis-remember code, leading to explanations ‘on the fly’. This may be a temporal issue, where other programmers have changed the code or the expert has not visited the code in a long time. Alternatively, it may be a comprehension issue, given that experts may have trouble remembering some code (Soloway and Ehrlich 1984). Future work will investigate whether these issues are accounted for in models of developers’ degree-of-knowledge of code (Fritz et al. 2010) and whether such models assist experts by recommending areas for review before conducting an onboarding session.

Acknowledgments This work is supported by Science Foundation Ireland grants 03/CE2/I303_1, 04/CE2/I303_1 and 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

Appendix 1

[SURVEY PREVIEW MODE] Participant background qu...

http://www.surveymonkey.com/s.aspx?PREVIEW_MO...

Participant background questionnaire

Exit this survey

1. Welcome

1 / 5

Thank you for your interest in my research.

This questionnaire asks you for some information on your background and your software engineering experience.

Next

Powered by **SurveyMonkey**
Check out our [sample surveys](#) and create your own now!

[SURVEY PREVIEW MODE] Participant background qu...

http://www.surveymonkey.com/s.aspx?PREVIEW_MO...

Participant background questionnaire

Exit this survey

2. Your background

2 / 5

*** 1. Your name:**

2. How much commercial software development experience do you have?

Years

Months

Number of different companies I worked at

3. How much other software development experience do you have? (e.g. voluntary open source contributions, coursework, personal projects)

Years

Months

Number of different projects worked on:

4. What is your highest qualification?

Level

Subject

5. Please list the programming languages that you have used.

Languages I am currently very familiar with

Languages I used extensively in the past

Languages I have used a little

Prev

Next

Powered by [SurveyMonkey](#)
Check out our [sample surveys](#) and create your own now!

[SURVEY PREVIEW MODE] Participant background qu...

http://www.surveymonkey.com/s.aspx?PREVIEW_MO...

Participant background questionnaire

Exit this survey

3. Learning and mentoring

3 / 5

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?

Prev

Next

Powered by [SurveyMonkey](#)
Check out our [sample surveys](#) and create your own now!

[SURVEY PREVIEW MODE] Participant background qu...

http://www.surveymonkey.com/s.aspx?PREVIEW_MO...

Participant background questionnaire

[Exit this survey](#)

4. Your current project

4 / 5

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:

	I don't know what it is	I have never used it	I have used it a little / used something similar	I have used it a lot	I am an expert (others ask for my help with it)
the programming languages used by this team	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
the team's codebase in general	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
the code that was under discussion during the recording	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
the source control system	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
the issue tracking system	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
the domain you are working in (e.g. healthcare, accounting)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
the team's products as an end user	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
design patterns (e.g. Observer, Strategy, Factory)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
mentoring other developers	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

[Prev](#)[Next](#)

Powered by [SurveyMonkey](#)
Check out our [sample surveys](#) and create your own now!

[SURVEY PREVIEW MODE] Participant background qu...

http://www.surveymonkey.com/s.aspx?PREVIEW_MO...

Participant background questionnaire Exit this survey

5. Thank you!

5 / 5

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)

Prev Done

Powered by **SurveyMonkey**
Check out our [sample surveys](#) and create your own now!

Appendix 2

The following questions comprise the set of standard questions used in the follow-up interviews of newcomers. Like the background questionnaire, the standard question evolved slightly 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 any 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 session have improved 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 3

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

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.

6. What actions by the expert are useful or useless to the newcomer?

	Very useful	Useful	Neither useful nor useless	Useless	Very useless
The expert goes through complex code line by line.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert describes the high level architecture.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert describes the clients.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert points out simple and complex areas of code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert points out costs (e.g. in terms of memory use, speed or file size).	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert names chunks of code (e.g. "the initialisation stuff").	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert goes through all the code line by line.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert points out similarities or commonly used idioms in the code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The expert points out the design patterns in use in the code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Please expand on your choices:

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.

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)

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.

8. If the newcomer drives, they are more engaged because they cannot "switch off".

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

9. If the newcomer drives, the session is slower and less fluid.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

10. If the newcomer drives, this improves their ability to find code independently after the session.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

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.

11. If the newcomer drives, it is hard for the expert to direct them (the expert may be "back seat driving").

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

12. If the newcomer drives, the expert can judge how well the newcomer is learning the system.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

13. If the newcomer drives, it is hard for the expert to navigate and locate code.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

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.

14. If the newcomer drives, the expert is prevented from going too quickly through the system.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

15. If the newcomer drives, they will find the session more confusing.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

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:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

17. If the expert points out temporary fixes in the code, the newcomer will find this:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

18. If the expert points out dead code, the newcomer will find this:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

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:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

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)

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.

21. If the session uses the newcomer's machine, it will be interrupted by missing configurations or data.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

22. If the session uses the newcomer's machine, the newcomer will not be blocked afterwards by environment problems.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

23. If the session uses the newcomer's machine, the explanation is not as smooth.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

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.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

25. If the code is well commented, the expert will find this:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

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.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

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.

27. Experts find the debugger a useful tool for explanations.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

28. Experts remember how to set up the newcomer's development environment.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

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:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

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:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

31. If the expert uses the debugger to step through code, the newcomer will find this:

Very useful Useful Neither useful nor useless Useless Very useless

☐ ☐ ☐ ☐ ☐

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.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

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.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

Please expand on your choice:

34. From the expert's point of view, the ideal session does not waste time on environment setup.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐ ☐ ☐ ☐ ☐

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.

35. From the newcomer's point of view, the ideal session engages the newcomer, even at the cost of frustration to the expert.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐
☐
☐
☐
☐

Please expand on your choice:

36. From the newcomer's point of view, the ideal session directly helps them to make early contributions.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐
☐
☐
☐
☐

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.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐
☐
☐
☐
☐

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.

38. From the newcomer's point of view, the ideal session includes help with environment setup.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐
☐
☐
☐
☐

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.

Strongly agree Agree Neither agree nor disagree Disagree Strongly disagree

☐
☐
☐
☐
☐

Please expand on your choice:

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?

	Easy to find out without expert help	Possible to find out without expert help	Usually need to ask an expert	Always need to ask an expert
How to use the editor or IDE.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The reasons for implementation choices.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
How to run the software as an end user.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The location of test data.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Whether documentation is up to date.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Whether this is the latest version of the source code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Workarounds for common problems.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
How to set up the development environment.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

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:

Very useful	Useful	Neither useful nor useless	Useless	Very useless
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Please expand on your choice:

42. If the expert gives their opinion on the qualities of information sources, the newcomer will find this:

Very useful	Useful	Neither useful nor useless	Useless	Very useless
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Please expand on your choice:

43. If the expert guesses answers to the newcomer's questions, the newcomer will find this:

Very useful	Useful	Neither useful nor useless	Useless	Very useless
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

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 the session:

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

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.

50. If the newcomer takes notes in the session, this is:

Very useful Useful Neither useful nor useless Useless Very useless

☐☐☐☐☐

Please expand on your choice:

51. If the newcomer takes breaks during the session, this is:

Very useful Useful Neither useful nor useless Useless Very useless

☐☐☐☐☐

Please expand on your choice:

52. In the early stages, the newcomer will find that diagrams with a lot of detail are:

Very useful Useful Neither useful nor useless Useless Very useless

☐☐☐☐☐

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

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Adair JG (1984) The Hawthorne effect: a reconsideration of the methodological artefact. *J Appl Psychol* 69(2): 334–345
- Adolph S, Hall W, Kruchten P (2011) Using grounded theory to study the experience of software development. *Empir Softw Eng* 16(4):487–513
- Afonso LM., Cerqueira RF de G and de Souza CS (2012), Evaluating application programming interfaces as communication artefacts. in *Proceedings of the Psychology of Programming Interest Group 2012*, pp 151–162
- Bass L (2007), *Software architecture in practice*. Pearson Education. ISBN: 0321815734
- Begel A and Simon B (2008a) Novice software developers, all over again. In *Proceedings of the Fourth international Workshop on Computing Education Research (ICER '08)*. ACM, New York, 3–14
- Begel A and Simon B (2008b), Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08)*. ACM, New York, 226–230
- Berlin L (1993), Beyond program understanding: A look at programming expertise in industry. In: *Empirical Studies of Programmers: Fifth Workshop*, pp 6–25
- Berlin LM and Jeffries R (1992), Consultants and apprentices: observations about learning and collaborative problem solving. In: *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work*, pp 130–137
- Boehm-Davis DA, Fox JE, Philips BH (1996) Techniques for exploring program comprehension. In: *Empirical studies of programmers: Sixth Workshop*, pp 3–37
- Brooks R (1983) Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18(6):543–554
- Buckley J, Mooney S, Rosik J and Ali N (2013), 'JITTAC: a just-in-time tool for architectural consistency'. In: *Proceedings of the 35th International Conference on Software Engineering*, pp 1291–1294
- Buckley J, O'Brien MP, Power N (2006) Empirically refining a model of programmers' information-seeking behavior during software maintenance. In *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group*, pp 168–182
- Buckley J, Rosik J, Herold S, Wasala A, Botterweck G and Exton C (2016), FLINTS: a tool for architectural-level modeling of features in software systems. In the proceedings of the 10th European Conference on Software Architecture Workshop. pp 14–22
- Charmaz K (2009) Shifting the grounds: Constructivist grounded theory methods. In: Morse JM, Stern PN, Corbin J, Bowers B, Charmaz K, Clarke AE (eds) *Developing grounded theory: The second generation*. Left Coast Press, Walnut Creek, pp 127–154
- Chen K and Rajlich V (2011), Case study of feature location using dependency graph, after 10 years. In: *Proceedings of the 18th International Conference on Program Comprehension*, pp 1–3
- Chen C, Zhang K and Itoh T (2012), Empirical evidence of tags supporting high-level awareness. *Cooperative Design, Visualization, and Engineering*, pp. 94–101
- Chochlov M, English M, Buckley J (2017) A historical, textual analysis approach to feature location. *Inf Softw Technol* 88:110–126
- Clements P, Garlan D, Bass L, Stafford J, Nord R, Ivers J, and Little R (2002), *Documenting software architectures: views and beyond*. Pearson Education. ISBN: 0201703726
- Corbin J and Strauss A (2008), *Basics of qualitative research: Techniques and procedures for developing Grounded Theory*. Sage Publications. ISBN: 141290644X
- Dagenais B, Ossher H, Bellamy RKE, Robillard MP and de Vries JP (2010), Moving into a new software project landscape. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp 275–284
- de Gialdino IV (2009), Ontological and Epistemological Foundations of Qualitative Research. at the Forum: Qualitative Social Research. 10(2), Article 30. Available at <http://www.qualitative-research.net/index.php/fqs/article/view/1299/3163> Accessed 30 Sept 2018

- Dekel U and Herbsleb J (2009a), Reading the documentation of invoked API functions in program comprehension, in IEEE 17th International Conference on Program Comprehension, pp 168–177
- Dekel U and Herbsleb JD (2009b), Improving API documentation usability with knowledge pushing, in Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp 320–330
- Denzin N (1983) Interpretive interactionism. In: Morgan G (ed) Beyond Method. Sage, California
- Detienne F (2002), Software design - cognitive aspects. Springer-Verlag. ISBN: 1852332530
- Detienne F, Soloway E (1990) An empirically-derived control structure for the process of program understanding. International Journal of Man-Machine Studies 33(3):323–342
- Dit B, Revelle M, Gethers M, Poshvanyk D (2011) 'Feature location in source code: a taxonomy and survey. J Softw Maint Evol Res Pract 25(1):53–95
- Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting Empirical Methods for Software Engineering Research. In: Shull F, Singer J, Sjøberg DIK (eds) Guide to Advanced Empirical Software Engineering. Springer, London
- Ellis D, Haugan M (1997) 'Modelling the information seeking patterns of engineers and research scientists in an industrial environment. J Doc 53(4):384–403
- Ericsson KA, Simon HA (1980) Verbal reports as data. Psychol Rev 87(3):215
- Fagerholm F, Johnson P, Guinea AS, Borenstein J, and Munch J (2013), Onboarding in Open Source Projects: A Preliminary Analysis. IEEE 8th International Conference on Global Software Engineering Workshops. pp 5–10
- Feigenspan J, Kästner C, Liebig J, Apel S, Hanenberg S (2012), Measuring programming experience. In 20th IEEE International Conference on Program Comprehension, pp. 73–82
- Fritz T, Ou J, Murphy GC, and Murphy-Hill E (2010), A degree-of-knowledge model to capture source code familiarity. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Vol. 1, pp 385–394
- Gamma E, Helm R, Johnson R and Vlissides J (1995), Design patterns: elements of reusable object-oriented software. Vol. 206, Addison-Wesley. ISBN: 0321700694
- Glaser BG, Strauss AL (1967) The discovery of Grounded Theory: Strategies for qualitative research. Aldine de Gruyter, Hawthorne ISBN: 0202302601
- Goncalves MK, de Souza CRB, Gonzalez VM (2011) Collaboration, information seeking and communication: An observational study of software developers' work practices. J Univ Comput Sci 17(14):1913–1930
- Gorton I (2006) Essential software architecture. Springer ISBN: 3–540–28713-2
- Hertzum M, Pejtersen AM (2000) 'The information-seeking practices of engineers: searching for documents as well as for people. Inf Process Manag 36(5):761–778
- Hoda R, Nobel J, Marshall S (2012) Developing a grounded theory to explain the practices of self-organizing agile teams. Empir Softw Eng 17(6):609–639
- Hunt A, Thomas D (2002) Software archaeology. IEEE Softw 19(2):20–22
- Jordan H, Rosik J, Herold S, Botterweck G, Buckley J (2015) Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads, in Proceedings of the IEEE 23rd International Conference on Program Comprehension, pp 174–177
- Johnson M, Senges M (2010) Learning to be a programmer in a complex organization: A case study on practice-based learning during the onboarding process at Google. J Work Learn 22(3):180–194. <https://doi.org/10.1108/13665621011028620> Accessed 17 Dec 2018
- Kelly T and Buckley J (2006), A context-aware analysis scheme for bloom's taxonomy, In: Proceedings of the 14th International Workshop on Program Comprehension, pp 275–284
- Kingrey KP (2002) Concepts of information seeking and their presence in the practical library literature. Libr Philos Pract (e-journal) Available at: <http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1035&context=libphilprac> Accessed 18 Aug 2016
- Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Softw Eng 32:971–987
- Ko AJ, DeLine R and Venolia G (2007), Information needs in collocated software development teams, In: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, pp 344–353
- Kuhlthau C (1988) Developing a Model of the Library Search Process: Investigation of Cognitive and Affective Aspects. Reference Quarterly 28(2):232–242
- Lakhotia A (1993) Understanding someone else's code: analysis of experiences. J Syst Softw 23(3):269–275
- LaToza TD, Venolia G and DeLine R (2006), Maintaining mental models: a study of developer work habits, In: Proceedings of the 28th International Conference on Software Engineering, pp 492–501
- Lawrance J, Burnett M, Bellamy R, Bogart C and Swart C (2010), Reactive information foraging for evolving goals, In: Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI '10, pp. 25–34

- Lawrance J, Bogart C, Burnett M, Bellamy R, Rector K, Fleming S (2013) How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans Softw Eng* 39:197–215
- Lee S, Kang S (2012) A study on guiding programmers code navigation with a graphical code recommender. In: Lee R (ed) *Software Engineering Research, Management and Applications*, Vol. 377 of *Studies in Computational Intelligence*. Springer, Berlin, pp 61–75
- Lethbridge T, Singer J, Forward A (2003) How software engineers use documentation: The state of the practice. *IEEE Softw* 20(6):35–39
- Lethbridge T, Sim S, Singer J (2005) Studying software engineers: Data collection techniques for software field studies. *Empir Softw Eng* 10(3):311–341
- Letovsky S (1987) Cognitive processes in program comprehension. *J Syst Softw* 7(4):325–339
- Lincoln YS, Guba EG (1985) Establishing trustworthiness. *Naturalistic Inquiry* 289:331
- Littman D, Pinto J, Letovsky S and Soloway E (1986), Mental models and software maintenance, In: *Empirical Studies of Programmers: First Workshop*, p. 80–93
- MacLeod L, Storey M-A, Bergen A (2015), Code, camera, action: how software developers document and share program knowledge using YouTube, In: *Proceedings of International Conference on Program Comprehension 2015*, pp 104–114
- Marchionini G (1997), *Information seeking in electronic environments*, Vol. 9, Cambridge University Press. ISBN: 0521586747
- Matroska (2013), Matroska media container. URL: <http://matroska.org/>. Accessed 19 June 2016
- McDonald DW and Ackerman MS (1998), Just talk to me: a field study of expertise location, In: *Proceedings of the 1998 ACM conference on Computer Supported Cooperative Work, CSCW '98*, pp 315–324
- McKeogh J and Exton C (2004), Eclipse plug-in to monitor programmer behaviour In: *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange*, pp 93–97
- Mockus A, Herbsleb JD (2002) Expertise browser: a quantitative approach to identifying expertise, in *Proceedings of the 24th International Conference on Software Engineering*, pp 503–512
- Muhr T (2013), *Atlas.ti v6*. URL: <http://www.atlasti.com>. Accessed 08 July 2016
- Murray AR (2006), *Discourse structure of software explanation: snapshot theory, cognitive patterns and grounded theory methods*, PhD thesis, University of Ottawa
- Murray A and Lethbridge T (2005a), Presenting micro-theories of program comprehension in pattern form, In: *Proceedings of the 13th International Workshop on Program Comprehension*, pp 45–54
- Murray A and Lethbridge TC (2005b), On generating cognitive patterns of software comprehension, In: *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research, CASCON '05*, pp 200–211
- Neville-Neil GV (2003) Code spelunking: Exploring cavernous code bases. *ACM Queue* 1(6):42–48
- Northrup DA (1997) *The problem of the self-report in survey research*. Institute for Social Research, York University
- O'Brien M (2007), *Evolving a model of the information-seeking behaviour of industrial programmers*, PhD thesis, University of Limerick
- O'Brien MP, Buckley J, Shaft TM (2004) Expectation-based, inference-based, and bottom-up software comprehension. *J Softw Maint Evol Res Pract* 16(6):427–447
- O'Brien M, Buckley J and Exton C (2005) Empirically studying software practitioners – bridging the gap between theory and practice'. In: *Proceedings of the 21 International Conference on Software Maintenance*, pp 433–442
- Pennington N (1987) Stimulus structures and mental representations in expert comprehension of computer programs. *Cogn Psychol* 19(3):295–341
- Perlow L (1999) The time famine: Toward a sociology of work time. *Adm Sci Q* 44(1):57–81
- Pirolli P, Card S (1999) Information foraging. *Psychol Rev* 104(4):643–675
- Poff MA (2003), *Pair programming to Facilitate the Training of Newly Hired Programmers*. Technical report, Florida Institute of Technology. URL: <http://hdl.handle.net/11141/116> Accessed 17 Dec 2018
- Ragavan SS, Kuttal SK, Hill C, Sarma A, Piorkowski D, and Burnett M (2016), Foraging Among an Overabundance of Similar Variants. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, pp 3509–3521. <https://doi.org/10.1145/2858036.2858469>
- Ragavan SS, Pandya B, Piorkowski D, Hill C, Kuttal SK, Sarma A, and Burnett M (2017), PFIS-V: Modeling Foraging Behavior in the Presence of Variants. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, pp 6232–6244. <https://doi.org/10.1145/3025453.3025818>
- Ratanotayanon S and Sim S (2006), When programmers don't ask, in *Proceedings of the 21st International Conference on Automated Software Engineering*, pp 9–16
- Razzaq A, Wasala A, Exton C, Buckley J (2019) The State of Empirical Evaluation in Static Feature Location. *ACM Trans Softw Eng Methodol (TOSEM)* 28(1)
- Riley J (1996), *Getting the most from your data*, 2nd edn, Technical and Education Services Ltd. ISBN: 0947885307

- Rist RS (1986), Plans in programming: definition, demonstration, and development, In First workshop on Empirical Studies of Programmers, pp 28–47
- Robillard MP, Coelho W, Murphy GC (2004) How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.* 30(12):889–903
- Rubin J and Chechik M (2013) A survey of feature location techniques, In: I. Reinhartz-Berger, Sturm A, Clark T, Cohen S, and Bettin J, (eds). *Domain engineering*, Springer, pp 29–58
- Seaman C (1999) Qualitative methods in empirical studies of software engineering. *IEEE Trans Softw Eng* 25(4): 557–572
- Seaman C (2002), The information gathering strategies of software maintainers. In: *Proceedings of the International Conference on Software Maintenance*, pp 141–149
- Shaft TM, Vessey I (1995) 'The relevance of application domain knowledge: the case of computer program comprehension. *Inf Syst Res* 6:286–299
- Sharif KY (2012), Open source programmers' information seeking, PhD thesis, University of Limerick
- Sharif KY, English M, Ali N, Exton C, Collins JJ, Buckley J (2015) An empirically-based characterization and quantification of information seeking through mailing lists during Open Source developers' software evolution. *Inf Softw Technol* 57:77–94
- Shaw M and Garlan D (1996), *Software architecture: perspectives on an emerging discipline*. Prentice Hall. ISBN: 0131829572
- Sheppard S, Curtis B, Milliman P, Love T (1979) Modern coding practices and programmer performance. *Computer* 12:41–49
- Shneiderman B, Mayer R (1979) Syntactic/semantic interactions in programmer behavior: A model and experimental results. *Int J Comput Inform Sci* 8(3):219–238
- Sillito J, Murphy G and De Volder K (2006), Questions Programmers ask during Software Evolution Tasks. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp 23–34
- Sillito J, Murphy G, De Volder K (2008) Asking and answering questions during a programming change task. *IEEE Trans Softw Eng* 34:434–451
- Sim S, Holt R (1998) The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. In: *Proceedings of the 1998 International Conference on Software Engineering*, pp 361–370
- Singer J (1998), Practices of software maintenance. In: *Proceedings of the International Conference on Software Maintenance, ICSM '98*, pp. 139–145
- Smith-Atakan S (2006), *Human Computer Interaction*. Thompson publishing. ISBN: 1–84480–454-2
- Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. *IEEE Trans Softw Eng* 10(5):595–609
- Starke J, Luce C and Sillito J (2009), Searching and skimming: An exploratory study. In: *Proceedings of the IEEE International Conference on Software Maintenance ICSM 2009*, pp. 157–166
- Stol, K-J, Ralph P, and Fitzgerald B (2016), Grounded Theory in Software Engineering Research. the 38th International Conference on Software Engineering, pp. 120–31
- Storey MA (2006) Theories, tools and research methods in program comprehension: past, present and future. *Softw Qual J* 14(3):187–208
- van Deursen A (2001). Program Comprehension Risks and Opportunities in Extreme Programming. [Proceedings Eighth Working Conference on Reverse Engineering](#). pp 176–185
- Van Maanen J, Schein EH (1979) Toward a theory of organizational socialization. *Res Organ Behav* 1:209–264
- VideoLAN (2013). VLC Media Player. URL: <http://www.videolan.org/vlc/>. Accessed 19th June 2016
- von Mayrhauser A and Vans AM (1993), From program comprehension to tool requirements for an industrial environment. In: *Proceedings of the IEEE Workshop on Program Comprehension*, pp 78–86
- von Mayrhauser A, Vans AM (1995a) Program understanding: Models and experiments. *Adv Comput* 40:1–38
- von Mayrhauser A, Vans AM (1995b) 'Industrial experience with an integrated code comprehension model. *Softw Eng J* 10(5):171–182
- von Mayrhauser A, Vans AM, Howe AE (1997), Program understanding behaviour during enhancement of large-scale software. In: *Journal of Software Maintenance: Research and Practice* 9 (5), pp 299–327
- Wiedenbeck S (1986) Beacons in computer program comprehension. *International Journal of Man-Machine Studies* 25:697–709
- Wilson TD (1981) On user studies and information needs. *J Doc* 37(1):3–15



Rebecca Yates was awarded a MComp degree in Computer Science from the University of Loughborough and a PhD in Software Engineering from the University of Limerick in 2014. She currently works as a software developer in Dublin, Ireland.



Norah Power received her MSc in Computing from the NIHE Limerick and her PhD in Software Engineering from Dublin City University. She lectured at the University of Limerick and supervised research in health informatics, requirements engineering and the skills of software professionals, with a particular focus on using qualitative research methods.



Jim Buckley was awarded an MSc degree in Computer Science from the University of Limerick and a PhD in Computer Science from the same University in 2002. He currently works as a senior lecturer in the Computer Science and Information Systems Department/Lero at the University of Limerick, Ireland. His main research interests focus on supporting software developers who are tasked with maintaining and evolving software systems. Thus, specific areas of interest include feature location, software comprehension and architectural analysis of such systems.