27th IEEE/ACM International Conference on Program Comprehension 2019
Preprint version

Live Programming and Software Evolution: Questions during a Programming Change Task

Juraj Kubelka
DCC, University of Chile, Chile

Romain Robbes
Free University of Bozen-Bolzano, Italy

Alexandre Bergel
DCC, University of Chile, Chile

***Abstract*—Several studies provide the questions developers ask during software evolution tasks, providing foundations for subsequent work. Nevertheless, none of them focus on Live Programming environments that gain in popularity as they are perceived to have a positive effect on programming tasks. Studying the impact of a Live Programming environment on software development activities is thus the goal of this study.**

In a partial replication of the study by Sillito *et al.*, we conducted 17 software evolution sessions in a Live Programming environment and report 1,161 developer questions asked during these sessions. We contrast our results with the results by Sillito *et al.*, focusing on the question occurrences, question complexity and what information participants used to gain a required knowledge. We report eight new questions and observe that the Live Programming facilities do have an impact on the way developers ask questions about source code and use tools to gain corresponding knowledge.

I. Introduction

Several foundational papers research the questions that developers ask during software evolution tasks. Sillito *et al.* identified 44 questions that developers asked during software change tasks, classified them in four categories, and documented how well-supported they are by state of the art tools [1]. Further studies identified additional aspects: LaToza *et al.* investigate reachability questions [2]; Ko *et al.* unveil developer day-to-day information needs [3]; and Duala-Ekoko *et al.* exhibit questions about unfamiliar codebase [4], to name a few. By documenting developer needs and how current approaches meet these needs, these studies provide valuable information to improve existing software evolution approaches, and to propose new ones.

Sillito's Ph.D. thesis [5] offers an extended account of the initial study and points to several possible follow-up studies, varying some dimensions of the study. One of these factors is the impact that available tools may have on the kind and frequency of the questions asked. To this aim, we performed a partial replication [6] of the study by Sillito *et al.* [1].

The principal variation is that our participants worked on tasks in a Live Programming environment. We chose it particularly because Live Programming environments have two important characteristics, together called *liveness* [7], that may greatly affect how developers work: (1) they always offer accessible evaluation of a source code, instead of the common edit-compile-run cycle, and as a consequence (2) they allow nearly instantaneous feedback to developers, instead of them having to wait for the program to recompile and run again before seeing any changes [8].

These characteristics may affect the type and frequency of developer questions, as well as the approaches they use to answer questions. It is especially important as these liveness features are nowadays experiencing a resurgence (details Section II-B)—while the Live Programming concepts date back to the LISP [9] and Smalltalk [10] environments of the 70's.

We present a critical opportunity to understand how liveness is used today in the growing Live Programming environments. Section II provides an essential background of the study by Sillito *et al.* that we partially replicate, followed by a context and a state of art in Live Programming and Pharo, and related work. Section III describes our prior studies, emphasizing the contribution of this extended work. Section IV then presents our research method. We observed 17 programming sessions carried out by 11 programmers. From a total of 13 hours of programming activity, both on unfamiliar and familiar codebases, we infer 1,161 developer question occurrences about source code asked by participants.

Section V answers our research question *"**What developer questions Pharo developers ask compared with the observations by Sillito* et al.*?"*** In the affirmative, we find that Pharo developers asked same questions as presented by Sillito *et al.*, varying in the frequency of individual questions. In addition, we document eight new questions.

Section VI answers our research question *"**What is the complexity of answering developer questions in Pharo considering involved sub-questions?"*** We find that questions about expanding focus points are simplest, while the other questions are complex, involving many sub-questions.

Sections VII and VIII answer our research question *"**How do Pharo developers use the development tools compared with the observations by Sillito* et al.*?"*** We find that Pharo developers tend to use runtime information, accessing objects easily and frequently.

Finally, we close the paper with threats to validity (Section IX), summary (Section X), and conclusion (Section XI).

II. Background and Related Work

A. The Study by Sillito et al.

This section describes the research by Sillito *et al.* [1] who conducted two studies. The first study was conducted in laboratory settings with nine computer science graduate students, working in pairs on tasks for 45 minutes. Participants worked on bug-fixing tasks, selected from an ArgoUML's issue tracking system. They used Java and the Eclipse IDE.

The second study was conducted in the industry. Sixteen developers worked alone for 30 minutes on a task of their own. In one case, there were two developers working together. Sillito *et al.* asked developers to choose realistic tasks. Participants used a variety of languages and tools: C, C++, C#, and Java languages and Emacs, VIM, Visual Studio, and Netbeans.

Sillito *et al.* then reported a list of 44 developer questions and classified them in four categories. They documented anecdotes observed while answering questions, and the development tool limitations. They propose several possible follow-up studies, suggesting that available tools might impact the kind and frequency of the developer questions [5]. To find out to what extent their results are valid, we partially replicate [6] the study with a different programming language and development environment: Pharo, a Live Programming environment.

The following two sections describe and motivate our choice to focus on Live Programming and the Pharo language and tools. We already describe this motivation in our previous work [11] that we reuse in the following two sections to cover fundamental background of this study.

B. Live Programming Environments

The reader wishing to experience Live Programming may visit the http://livecoder.net web site for a JavaScript example.

Liveness. Live Programming gives developers nearly instantaneous feedback from their programs through always accessible evaluation [8]. Tanimoto defines four liveness levels [7]. At level one, an application has to be manually restarted to obtain a feedback on a code change. At level two, semantic feedback is provided on-demand by interactive interpreters (Read-Eval-Print loops). At level three, incremental feedback is provided after each edit operation. Finally, at level four, changes are applied to a running system without explicitly initiating the application under the change. Those levels provide different experiences covering auto-testing solutions, Read-Eval-Print loops (REPLs), and systems where development tools and running applications share the same running environment [12]. While Live Programming dates back decades, with support in LISP [9], Smalltalk [10], Self [13], or Squeak [14] a renewed interest has been seen in recent years.

Academia. Two research events, the LIVE programming workshop [15] and the International Conference on Live Coding [16], held their third edition in 2017. Live Programming languages and tools were presented, most of them as experimental. As we focus on Live Programming in practice and due to space constraints, we provide only compact coverage of a handful of research. Tanimoto presented VIVA, a visual language for image processing [17], Burnett *et al.* implemented level four liveness in a visual language [18]. Recent solutions include languages such as McDirmid's SuperGlue [19], Jonathan Edward's Subtext [20], and Glitch [21]. Microsoft TouchDevelop obtained Live Programming support [22]. DeLine *et al.* introduced Tempe, a Live Programming environment for data analysis [23].

Industry. Live Programming is integrated in many frameworks and tools, particularly on the web. The Google Chrome web development tools [24] can change web pages and JavaScript code without a webpage reload. Similarly, popular JavaScript frameworks including Facebook's Reacts.js support Live Programming [25]. Our previous study shows that those Live Programming features are partially used by developers [11]. Java applications can replace parts at runtime through the Java Platform Debugger Architecture [26]. Apple Xcode supports interactive Swift playgrounds [27]. Microsoft Visual Studio 2015 allows REPLs interactive programming for C# and F# [28], [29]. Finally, Microsoft acquired CodeConnect, a startup company which developed a Live Programming extension for Visual Studio [30].

C. Live Programming in Pharo

Why Pharo? Pharo [31] is a programming language and IDE derived from Squeak [14], a dialect of Smalltalk-80 [10]. Pharo itself can update any part of an application at runtime. It therefore supports level four of liveness defined by Tanimoto (which is not the case with other modern IDEs).

As a descendant of Smalltalk-80, liveness has been present in Pharo for over thirty years and there are two reasons that we chose Pharo for this study. First, liveness was designed in the language and tools since its inception and its support is mature. Second, liveness is also part of the development community culture, both in industry and academia. Pharo users are introduced to liveness since the very beginning and practitioners use it naturally as part of their daily tasks. It is the main driver of our choice, as we wanted to know how developers, familiar with liveness, use it in practice. Next, we contrast the main liveness features of Pharo with Eclipse, a traditional IDE with *very limited* liveness support.

Code evaluation. Pharo includes a compiler as a user-accessible object which can evaluate *any piece of text*. Evaluated code can be printed out, inspected, debugged, or profiled. This has many consequences in the community culture, *e.g.,* users can encounter executable examples embedded in class and method comments, select and execute the examples, and observe results. Pharo also allows compiling and updating any method even while an application is running. Eclipse only allows executing applications in normal and debug mode, and executing test cases.

Accessing objects. The object inspector tool displays object states [32]. Developers can navigate the object graph, modify instance variables, evaluate code snippets, browse its source code, and invoke inspectors from almost anywhere. Users can open multiple inspectors at the same time, and keep them for arbitrarily long time periods. Eclipse offers a limited object inspector in its debugging perspective, with limited possibility to navigate object graphs and to change values, and it is dismissed as soon as developers leaves the debugger.

Playground. Arbitrary code snippets may be written and executed in a playground. Results may be explored in embedded inspectors. Several playgrounds can be opened simultaneously and indefinitely, which are absent in Eclipse.

Debugger. The Pharo debugger consists of a stack trace at the top, a source code editor in the middle, and an embedded

object inspector at the bottom. Users can manipulate variable values, edit source code, and observe effects of these changes. Several debuggers can be opened at the same time. It allows for easy comparison of two execution scenarios. Eclipse debugger allows code evaluation, with a *limited* possibilities to explore its results. Eclipse does *not* support multiple opened debuggers and changing source code at run-time is also limited.

D. Studies on Developer Information Needs

Several studies focus on developer information needs and provide a structured list of developer questions.

Ko *et al.* identify 21 questions about interaction with a codebase and co-workers [3]. They categorize questions into seven categories: writing code, submitting a change, triaging bugs, reproducing a failure, understanding execution behavior, reasoning about design, and maintaining awareness.

LaToza *et al.* focus on answering reachability questions and provide 12 questions rated by difficulty and frequency [2]. They identify seven developer activities during which developers spent the most of the time debugging or proposing changes and investigating their consequences. They conclude that the longest debugging and implication activities were associated with reachability questions.

LaToza *et al.*, in another study, surveyed professional software developers and provide 94 hard-to-answer developer questions in 21 categories [33]. The list covers questions about (past or future) changes, types (classes, methods, functions, *etc.*), and type relationships.

Fritz *et al.* present 78 developer questions with lack of tool support that involve information integration from codebases, work items, change sets, teams, comments, and Internet [34].

De Alwis *et al.* extract 36 developer questions from literature, blogs, and their own experience [35]. They state that answering most of the questions involves using a variety of tools, forcing a programmer to piece together tool results to answer the initial questions.

LaToza *et al.* present developer activities and the tool support level for them [36]. Developers spend half of their time fixing bugs, 36% writing new features, and the rest of their time making code more maintainable. They present developer difficulties for each activity and highlight solutions.

To our knowledge there is *no* similar study performed on Pharo or any another Live Programming environment. The research works presented above are conducted in Java, C, C++, C#, or Visual Basic and their corresponding IDEs, *e.g.,* Emacs, VIM, Eclipse, Microsoft Visual Studio.

III. Prior Studies

This section describes our previous workshop paper [37] (Study A) and a follow-up of our study [11] (Study B).

In the Study A, we analyze six programming sessions. We report that question occurrences are similar on unfamiliar codebase and differ on familiar codebase comparing it with results by Sillito *et al.* Study A does *not* provide in-depth understanding about its result: question dependencies, question complexity, and tool usage are not studied, although it is crucial to understand the development workflow. Study A comes up with eight additional developer questions.

TABLE I
Participant information.

Participant Id	Programming experience in Any language [years]	Programming experience in Smalltalk [years]	Current Position	Conducted Sessions
P1	5.5	3	Professional, Ph.D. Student	S1
P2	15	11	Professor	S2
P3	5	1.5	Professional	S3
P4	20	13	Professional, Professor	S8, S11
P5	7	0.5	Professional, Master Student	S9, S12
P6	22	16	Professor	S10, S13
P7	10	6	Professional, Ph.D. Student	S4
P8	4	2	Bachelor Student	S5, S14
P9	7	3	Ph.D. Student	S6, S15
P10	3	0.5	Professional	S16
P11	5	3	Ph.D. Student	S7, S17

In the Study B, we analyze if and how developers use Live Programming features. We report that Live Programming features and tools are used extensively by Pharo users. We describe several usages of Live Programming and contrast them against traditional programming approaches. Our overall finding is that simple approaches were favored and combined (confirmed by 190 survey responses, 68% from industry).

This paper presents results of 17 programming sessions of the Study B (including 6 sessions of the Study A). We provide more understanding why developer question occurrences vary contrasting it with Sillito *et al.* We look closer at how questions are linked and what their complexity is. Finally, we contrast tool usage to answer developer questions in our study and the study by Sillito *et al.*

IV. Research Method

This section describes the settings of our exploratory study, previously described in our prior studies (see Section III).

Participants. We employed eleven male participants, including students, academic staff, and professional developers from distinct small local companies. Participants' programming experience range from 5 to 22 years, with a median of 6 years. Experience in Pharo and Smalltalk ranges from 0.5 to 16 years, with a median of 3 years. There was 1 bachelor student, 2 Ph.D. students, 2 professors, 6 professional developers (four with mixed positions). All but P2 and P5 participants also contribute to free–software projects. See details in Table I.

Tasks. We conducted 17 programming sessions, consisting of 10 sessions (S1–S10) on *unfamiliar codebase*, and 7 sessions (S11–S17) on *familiar codebase*. Some participants did not get involved in familiar codebase sessions as they reported not being a software project author that time.

We classified a session as *familiar* if a participant was one of the authors of the codebase. Participants chose a programming task of his project on which he can work during 40 minutes, understanding that it was not important to finish the task during the session. In the case of *unfamiliar* sessions, participants had little or no knowledge of a codebase beforehand. We provided them a task description (available at [38]) and a prepared Pharo programming environment related to a particular bug-fixing or enhancement task of Pharo and Roassal [**?**].

Study Setting. Participants conducted the study using Pharo version 3. They could use all IDE features and documentation resources. We advised them to proceed with the tasks as usual. We configured devices and explained the session procedure before each session. We asked participants to verbalize their thoughts during each session. Participants did not get any prior training for our study. After about 40 minutes, we informed participants that they could finish whenever they wanted.

Data Collection and Transcription. We used two data collection techniques: screen captured audio and videos (13 hours in total) and user interactions [**?**]. We described participant thoughts and actions they were performing on their screens. We developed tools to attach timestamps to the transcripts and to navigate back to the videos as needed during the analysis. We used those tools to sanitize transcripts and to minimize possible analysis errors. These data, recordings, and tools are available at [**?**], [**?**], [38].

Questions Extraction. To extract the questions in our study, we first identified the *concrete questions*. In this phase, we went through the audio and video records and produced a semi-structured transcript. We verbalized actions in the transcript as concrete questions annotated by the `Q` symbol, *e.g.,* Q*"How is the background created for the parent menu?"* Some questions were explicit, *e.g.,* while P3 was observing a particular method, he asked *"Why does it not do the same things at the same time?"*. Other questions were inferred from the user actions, *e.g.,* P1 jumped from the code where the `TRMouseClick` class was used and observed its class definition and its methods. This yielded the question *"What are the parts of TRMouseClick?"*

After identifying concrete questions we then synthesized *generic questions* that extract the specifics of a given task. We include generic questions in the transcript annotated by the `GQ` symbol, *e.g.,* GQ*"(23) How is this feature or concern (object ownership, UI control, etc.) implemented?"*, as shown in the following transcript excerpt:

- 06:35-08:08 P2 asks Q*"How is the background created for the parent menu?"* GQ*"(23) How is this feature or concern (object ownership, UI control, etc.) implemented?"*
  - 06:39-06:44 P2 goes to method `createParentMenu:background:` observing the implementation where P2 sees another method Q*"What does the method look like?"* GQ*"(17) What does the declaration or definition of this look like?"* which creates background
  - 06:44-07:05 then P2 asks Q*"Why does it not do the same things [parent menu label and background color] at the same time?"* GQ*"(25) What is the behavior that these types provide together and how is it distributed over the types?"*

Some of the concrete questions we identified are not conveniently mappable to the list of questions presented by Sillito *et al.*, *e.g.,* Q*"Why does the test case fail?"* If none of the questions proposed by Sillito *et al.* match a question in our study, we abstract the question; for example we map the question *"Is the `R3CubeShapeclass` tested?"* to the generic question *"(e6) Is this entity or feature tested?"* New questions are added to one of the categories identified by Sillito *et al.*

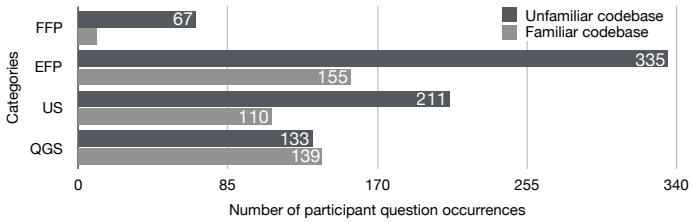

Fig. 1. Question occurrences, including all questions mentioned in Table II.

The transcript was performed by the first author. When the author was unsure of the transcription, all authors held discussions. To minimize biases in the interpretation, we built tools that allowed us to move between reports, transcripts, and audio and video recordings.

V. Results: Developer Questions

We collect 1,161 question occurrences from 17 programming sessions, in 13 hours of videos. On average, we report one question every 36 seconds on unfamiliar and every 47 seconds on familiar codebase.

Table II presents a list of all the questions, grouped by the categories identified by Sillito *et al.* The list includes new questions (discussed later), not identified by Sillito *et al.*, in corresponding categories. The second column group compares tool usage (explained later) in our study and the study by Sillito *et al.* The third column group gives an aggregate count of question occurrences in our two type sessions, compared to Sillito *et al.* Since the numbers are not directly comparable, we use the cell background to show the relative frequency of the questions: darker backgrounds indicate more frequent questions in corresponding sessions (columns). Finally, the last two column groups show detailed information about our sessions, again using background color to encode frequency.

Figure 1 shows the distribution of the question occurrences among categories for unfamiliar and familiar codebases. We observe the least questions in the FFP category (in particular in familiar codebase sessions) and most questions in the EFP category. EFP and US questions occurred about twice more in unfamiliar codebase sessions than in familiar. We detail each category in the following sections.

A. Finding Focus Points (FFP)

Most of our participants asked questions about finding initial points in their codebase that were relevant to the task. These questions were about finding types that correspond to domain concepts. Participants searched for classes that correspond to graphical widgets, *e.g., "I want to know where the window is opened and put a breakpoint there"* (P11 in S7), *"Who is responsible for executing this dialog?"* (P5 in S9), *"I want to know where 'History Navigator' string is used, in order to*

Question Types per Category	Tool usage By Us	Tool usage By Sillito	Question occurrences in Our sessions Unf.	Our sessions Fam.	Sillito ses. Unf.	Sillito ses. Fam.	Our sessions on unfamiliar code S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	Our sessions on familiar code S11	S12	S13	S14	S15	S16	S17
Finding Focus Points (FFP)																							
(1) Which type represents this domain concept or this UI element or action?	s/d	s	11	3	8				1		4	1	2		2	1		2	1				
(2) Where in the code is the text in this error message or UI element?	S	s	1		4										1								
(3) Where is there any code involved in the implementation of this behavior?	s	s/d	23		10	2	3	6	3		1	2	3		4	1							
(4) Is there a precedent or exemplar for this?	s	s	14	4	4	4	1		1	4		1	3	3		1		2	1	1			
(5) Is there an entity named something like this in that unit (project, package, or class, say)?	S	s	18	4	11	1	4					1	4	2	7			1	1	1			1
Total in the category			67	11	37	7	8	6	5	4	5	5	12	5	14	3		5	3	2			1
Expanding Focus Points (EFP)																							
Types and Static Structure																							
(6) What are the parts of this type?	S	s	60	31	11	1	11	5	1	2	3	13	6	6	7	6	3	4	1	5	12	2	4
(7) Which types is this type a part of?	s	s	3	2	2			1					1		1			1					1
(8) Where does this type fit in the type hierarchy?	S	s	10	1	10		3	1			1		3	1	1					1			
(9) Does this type have any siblings in the type hierarchy?	S	s		1	2												1						
(10) Where is this field declared in the type hierarchy?	S	s		2	2												2						
(11) Who implements this interface or these abstract methods?	S	s	8		5		2	1	4	1													
Extra Questions on Types and Static Structure																							
(e1) Where is the method defined in the type hierarchy?	S	-	3	1	-	-	1			2										1			
(e2) What are the correct arguments names to this method?	S	-		8	-	-											4		3	1			
(e3) Which abstract methods should be implemented to this type?	S	-		2	-	-													2				
Incoming Connections																							
(12) Where is this method called or type referenced?	S	s	41	23	33	2	7	2	1	9	3	7	2	4	2	4	2	1	7	1	2		10
(13) When during the execution is this method called?	S	d	1		4	1	1																
(14) Where are instances of this class created?	D	s	4		8			1			1		1		1								
(15) Where is this variable or data structure being accessed?	S	s	13	2	8	3	3	2	3	3				2						2			
(16) What data can we access from this object?	S	s	1		1	1					1												
Outgoing Connections																							
(17) What does the declaration or definition of this look like?	S	s	140	72	13	5	8	13	16	10	15	21	14	22	10	11	18	1	15	3	6	5	24
(18) What are the arguments to this function?	s	s	4	1	10		3		1											1			
(19) What are the values of these arguments at runtime?	D	d	22	4	4	1			6	6	5	5								4			
(20) What data is being modified in this code?	s	s	3	1	2				3										1				
Extra Questions on Outgoing Connections																							
(e4) What method implementation corresponds to my question?	s	-	4		-	-	1	2						1									
(e5) What is the variable type or what is the method's return type at runtime?	s/d	-	18	4	-	-	2	3	3			2	2	3	1	2		1		1		1	1
Total in the category			335	155	115	14	42	31	38	33	29	48	29	39	23	23	30	8	29	20	20	8	40
Understanding a Subgraph (US)																							
Behavior																							
(21) How are instances of these types created and assembled?	s/d	s	8	1	9		3		2							3							1
(22) How are these types or objects related? (whole-part)	S	s	3	1	3		1	1	1									1					
(23) How is this feature or concern (object ownership, UI control, etc.) implemented?	s	s	30	12	12	3	6	3	7	4	1	4		1	1	3		2	1		6		3
(24) What in this structure distinguishes these cases?	-	s			2	1																	
(25) What is the behavior that these types provide together and how is it distributed over the types?	s/d	s	26	3	7	1	6	5	7	1	1	1	1		2	2				1			2
(26) What is the "correct" way to use or access this data structure?	s/d	s	8	11	3	3					1	5		2				4			1	5	1
(27) How does this data structure look at runtime?	D	d	72	34	3	3	2	3	3	8	2	1	6	3	15	29	4	11	2		11	6	
Data and Control Flow																							
(28) How can data be passed to (or accessed at) this point in the code?	s/d	d	10	5	4	1		3		1	2	3	1								5		
(29) How is control getting (from here to) here?	D	d	5		2			1			1	2			1								
(30) Why is not control reaching this point in the code?	s/d	d	3	1	6	2					1	1		1					1				
(31) Which execution path is being taken in this case?	D	d	13	3	7	2			1		1		1	1	7	2			2			1	
(32) Under what circumstances is this method called or exception thrown?	D	d	3	1	8				2			1					1						
(33) What parts of this data structure are accessed in this code?	-	s			3																		
Extra Questions on Data and Control Flow																							
(e8) What does the failure look like?	d	-	30	38	-	-		1	5	7	3	1	2	7	2	2	11	3	2	8	5	2	7
Total in the category			211	110	69	16	18	17	28	21	13	19	11	15	28	41	16	21	8	9	28	14	14
Questions over Groups of Subgraphs (QGS)																							
Comparing or Contrasting Groups																							
(34) How does the system behavior vary over these types or cases?	d	s/d	23	21	2	1	2	2	2	6	2	1		4		4	5	2			1		13
(35) What are the differences between these files or types?	S	s	1	9	1	8						1					5	1	1			2	
(36) What is the difference between these similar parts of the code (e.g., between sets of methods)?	s	s	2	2	3	4			1	1								1			1		
(37) What is the mapping between these UI types and these model types?	-	s			4																		
Change Impact																							
(38) Where should this branch be inserted or how should this case be handled?	s	s	36	24	5	2	1	2	8	1	3	2	5	11	1	2	4	4	6	5	4	1	
(39) Where in the UI should this functionality be added?	s	s	1		4	2							1										
(40) To move this feature into this code, what else needs to be moved?	S	s	2			2							2										
(41) How can we know that this object has been created and initialized correctly?	s/d	s	1	5	2				1								4			1			
(42) What will be (or has been) the direct impact of this change?	d	s/d	28	16	7	15		1	4	2	2	6	3	9		1	1	2	11		1		1
(43) What will the total impact of this change be?	d	s/d	4	1		9			1					2		1							1
(44) Will this completely solve the problem or provide the enhancement?	D	d	22	6	3	2		3	7	1	5	1	3	2				3			2	1	
Extra Questions on Change Impact																							
(e6) Is this entity or feature tested?	S	-	2	3	-	-				2							2					1	
(e7) Do the test cases pass?	D		11	52	-	-				11							21	1		15	15		
Total in the category			133	139	31	45	3	8	24	24	12	11	14	28	1	8	42	14	18	21	24	5	15
Total	-		746	415	252	82	71	62	95	82	59	83	66	87	66	75	88	48	58	52	72	27	70
Tool Usage Legend: S = strong static, s = static, s/d = static and dynamic, d = dynamic, D = strong dynamic tool usage.																							

TABLE II
TOOL USAGE AND QUESTION OCCURRENCES OBSERVED IN THIS STUDY AND THE STUDY BY SILLITO *et al.*

find out a class that implements it." (P8 in S5), and *"Does the number 46 represent a dot character?"* (P6 in S10).

Participants in sessions on unfamiliar codebase usually asked the questions in this category at the beginning of each session. Forty-eight percent of questions in this category were asked in the first quarter of each session, and 22% in the second quarter, see Figure 2. In sessions on familiar codebase, we observe 11 questions that were asked mostly in the second part of each session, with the distribution of 27%, 0%, 36%, and 36% for each quarter. Sillito *et al.* also observed that FFP questions were asked at the beginning of sessions and also when participants began to examine new codebase parts.

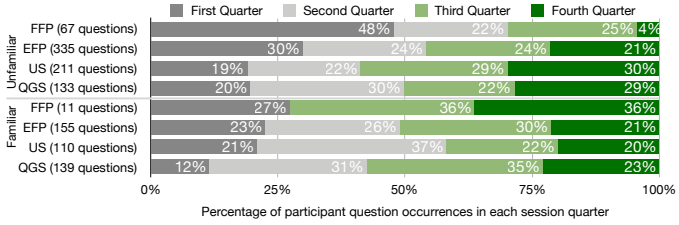

Fig. 2. Question occurrences asked during four session quarters.

B. Expanding Focus Points (EFP)

Questions in this category are about building from an entity, *e.g.,* class or method, and exploring other relevant entities. Participants often answered these questions by exploring relationships, *e.g.,* "*What instance variables does the 'BPVariable' class have?*" (P5 in S5), "*Who are senders of 'rawmenu' method?*" (P2 in S2), "*Is the 'R3CubeShape' class referenced in a test?*" (P4 in S11), and "*What do the error handling methods look like?*" (P6 in S13).

Questions in this category are more frequent for unfamiliar sessions. Sillito *et al.* found even more prominent differences. Comparing question occurrences, questions *"(6) What are the parts of this type?"*, *"(12) Where is this method called or type referenced?"*, and *"(17) What does the declaration or definition of this look like?"* are the most frequent questions in this category. Those questions involve basic source code observations, *e.g.,* class and method definitions and references to the rest of a codebase. Question 17 is the most frequent in our study. Questions 12, 17, and 6 are also most frequent questions in this category in the study by Sillito *et al.* Question 12 is the most frequent in their study.

C. Understanding a Subgraph (US)

Questions in this category are about understanding of concepts that involve several entities and relationships. Participants were often interested how a code works and how its behavior is implemented, *e.g.,* "*How does it [a click in a text editor] work now in Nautilus [application]?*" (P11 in S7), "*Which part of code is responsible for this behavior [box placement]?*" (P2 in S2), and "*What [...] filename [is] related to this variable [...]?*" (P5 in S12).

Question *"(27) How does this data structure look at runtime?"* is the most prevalent questions in this category. Participants asked it predominantly in unfamiliar sessions as they knew less about the corresponding codebases. Another frequent question *"(e8) What does the failure look like?"* in this category is our additional question discussed in Section V-E. Sillito *et al.* observe question *"(23) How is this feature or concern (object ownership, UI control, etc.) implemented?"* as most frequent.

D. Questions over Groups of Subgraphs (QGS)

While questions in the previous section involve understanding of a subgraph, questions in this category involve understanding the interaction between multiple subgraphs or the subgraph and the rest of the system. Participants asked the questions when they investigated how programs behave in different scenarios, *e.g.,* "*How fast is this widget?*" (P11 in S17), "*Why is the 'anEvent' not keystroke?*" (P6 in S10), "*How is the 'position' method implemented [in two different clases]?*" (P7 in S4), and "*Where can I set the color?*" (P3 in S3).

Questions *"(34) How does the system behavior vary over these types or cases?"*, *"(38) Where should this branch be inserted or how should this case be handled?"*, *"(42) What will be (or has been) the direct impact of this change?"*, and *"(e7) Do the test cases pass?"* (a new question discusses later) are the most prevalent questions in this category. Sillito *et al.* report questions 42, 43 as most frequent in this category and among all questions on familiar codebase.

E. Additional Questions

This section discusses aggregated questions that are not present in the study by Sillito *et al.*

1) On Dynamically Typed Aspects of Pharo: In this section we discuss the following questions:

- *"(e1) Where is the method defined in the type hierarchy?"*
- *"(e2) What are the correct argument names to this method?"*
- *"(e3) Which abstract methods should be implemented to this type?"*
- *"(e4) What method implementation corresponds to my question?"*
- *"(e5) What is the variable type or what is the method's return type at runtime?"*

Polymorphism. In certain cases, answering question *"(17) What does the declaration or definition of this look like?"* needed to be divided into sub-questions. Participant P2 first had to determine to whom the message is sent by asking question *"(e5) What is the variable type or what is the method's return type at runtime?"* It was answered by putting a breakpoint into a specific method and subsequently observed in a debugger.

A static observation of a particular codebase was also a common practice. Participant P1 estimated the original question 17 by asking *"(e4) What method implementation corresponds to my question?"* supposing that it should be a class corresponding to the same package that he manipulated. Since he did not find the expected class, he browsed the class definition asking *"(e1) Where is the method defined in the type hierarchy?"* and he searched the method in the following superclasses where he found the answer.

In the statically typed languages, *e.g.,* Java, question 17 is answered by a direct navigation from a calling method to a particular definition. In Pharo, it is necessary to perform extra steps (questions e1, e4, and e5) to achieve the information. Sillito *et al.* also observe navigation difficulties (in Java tasks) when polymorphism, inheritance events, and reflection are involved, making results noisy and hard to interpret.

Implementation. The Pharo language has no special symbol distinguishing abstract class or method declaration. Instead, a dedicated method call is used in the definition of a particular "abstract" method. If a developer forgets to override the method, an exception is raised. Therefore, at the time of defining a new class, participant P6 has checked the methods in the superclass, asking question *"(e3) Which abstract methods should be implemented to this type?"*

Participant P4 was in the opposite situation. He formed a method that is a part of an abstract programming interface (API). Since argument names are an important API guideline in a dynamically typed language, he was interested in what the argument names are in other methods: question *"(e2) What*

are the correct argument names to this method?" The name was `aValueOrASymbolOrAOneArgBlock` indicating that values can be basic ones (*e.g.,* numbers and strings), a symbol (a specialized string), or a one-argument lambda function.

2) On Test Cases: Here we discuss the following questions:

- *"(e6) Is this entity or feature tested?"*
- *"(e7) Do the test cases pass?"*
- *"(e8) What does the failure look like?"*

Participant P4 began his work writing test cases. First, he wondered whether a particular scenario is tested, *i.e.,* question *"(e6) Is this entity or feature tested?"* He asked sub-question *"(12) Where is this method called or type referenced?"* In that particular case he found it difficult to answer the question e6 and noted that *"this is not worth wasting time over ... writing a test should be easy"* and he wrote a new one. Later, when he was fixing the test cases affected by his changes, he found tests similar to those he wrote at the beginning.

Questions *"(e7) Do the test cases pass?"* and *"(e8) What does the failure look like?"* are recognized by Sillito *et al.* Question e7 could be mapped to *"(42) What will be (or has been) the direct impact of this change?"* or *"(44) Will this completely solve the problem or provide the enhancement?"* Question e8 could be mapped to *"(29) How is control getting (from here to) here?"*, *"(30) Why is control not reaching this point in the code?"*, or *"(32) Under what circumstances is this method called or exception thrown?"* Since it was difficult to identify specific questions, we used a more general form.

Conclusion. In this section, we report question occurrences contrasting them with the study by Sillito *et al.* Questions 17 (EFP), 27 (US), and e7 (QGS) are prevalent in our study, while questions 12 (EFP), 42 (QGS), and 43 (QGS) are asked frequently in the study by Sillito *et al.* New questions arise to explicitly document: (i) the source code navigation difficulties; and (ii) impediments to infer Sillito *et al.* questions.

Observation 1. Question "(17) What does the declaration or definition of this look like?" is the most asked question about source code. Question "(27) How does this data structure look at runtime?" is the most asked question about runtime.

VI. Results: Developer Question Complexity

Sub-questions. Session transcripts are structured to root questions, initial participant questions, and sub-questions that participants used to answer the root questions. Questions are therefore structured as a tree-graphs. In the following analysis, we classify questions to four groups: (1) root questions without sub-questions, (2) root questions with sub-questions, (3) inner-graph questions that have parent- and sub- questions, and (4) leave questions that have parent questions. Figure 3 shows this classification around four question categories defined by Sillito *et al.* For easy reading, we put questions with sub-questions to the left, and questions without sub-questions to the right.

We observe that 75% (876 out of 1,161) of questions do not include sub-questions (the right side in Figure 3). Those questions are organized around categories as: 4% FFP (Finding Focus Points), 50% EFP (Expanding Focus Point), 24% US (Understanding a Subgraph), and 22% QGS (Questions over Groups of Subgraphs). As a first approximation, we conclude that EFP questions are *simple* questions, understanding that they do not require sub-questions to answer them. QGS and US questions require more steps to answer. FFP questions were the most complex to answer (in terms of sub-questions).

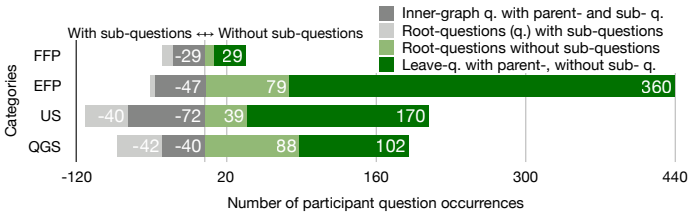

Fig. 3. Question and sub-question occurrences classified to root questions with and without sub-questions, inner-graph questions, and leave questions.

Looking at each question category separately, we observe similar results. Figure 3 shows that 49% FFP, 90% EFP, 65% US, and 70% QGS questions do not include sub-questions. Participants were therefore able to answer EFP questions predominantly directly (without involving sub-questions). EFP questions are then followed by US and GQS categories. FFP questions are most complex considering this ranking. We do not analyze if questions were answered (correctly or incorrectly) or abandoned as this requires further effort and is out of the scope of this study.

Root questions. Figure 4 closely shows how root questions with sub-questions (97 out of 312 root questions) were answered. Each question category is divided into four sub-question categories. To ease the reading, we put US and QGS categories to the left (less satisfactory tool support) and FFP and EFP categories to the right (satisfactory tool support), following the observations by Sillito *et al.* We observe that most categories are interconnected and participant picked sub-questions from all categories to answer root questions.

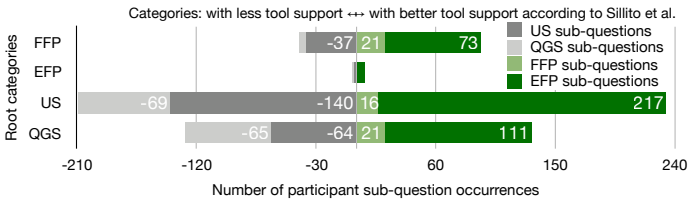

Fig. 4. Root question occurrences classified to sub-question categories.

Participants used 15% FFP, 53% EFP, 27% US, and 4% QGS sub-questions to answer FFP root questions. EFP root questions rarely required sub-questions (9 sub-questions in total). US root questions required 4% FFP, 49% EFP, 32% US, and 16% QGS sub-questions. Finally, QGS root questions involved 8% FFP, 43% EFP, 25% US, and 25% QGS sub-questions.

We observe that participants used EFP sub-questions often to answer root questions, followed by US and QGS sub-

questions. It supports our previous conclusion that EFP questions are basic instruments to understand a codebase. FFP sub-questions were least used as FFP questions are mainly asked when developers do not know anything about a codebase. Participants used FFP questions predominantly at the beginning of each session as we report in Section V-A.

Except for frequent use of EFP sub-questions, participants tend to have US and QGS sub-questions too. The proportion of QGS sub-questions is highest for QGS root questions, followed by US and by FFP root questions. Similarly, the proportion of US sub-questions is higher for US root questions, followed by FFP and QGS root questions.

Unfamiliar vs. familiar codebase. Figure 5 considers root questions (312 in total). It shows the number of sub-questions for root questions in each category, contrasting unfamiliar and familiar codebase sessions. It indicates that US and QGS root questions are more complex to answer (considering the number of required sub-questions) on unfamiliar codebase. There are about three times more sub-questions (mean values) on unfamiliar codebase. We cannot really contrast FFP root questions as we observe only two FFP root questions on familiar codebase. EFP root questions have similar complexity in terms of number of sub-questions (almost zero sub-questions). We observe that US and QGS root questions are most difficult to answer (in terms of involved sub-questions).

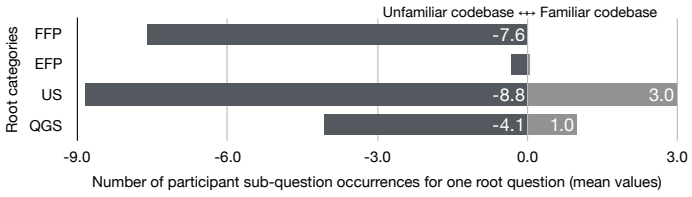

Fig. 5. Sub-questions occurrences for one root question (mean values).

Conclusion. In this section, we present question complexity based on the number of sub-questions involved answering root questions. We conclude that QGS, US, and FFP questions are the most complex to answer. EFP questions are the most simple to answer and were involved most in answering questions of other categories. QGS and US questions are more complex in unfamiliar codebase sessions.

Observation 2. QGS, US, and FFP questions are complex. EFP questions are simple and are used to answer other questions.

VII. Results: Tool Usage Answering Questions

Tool Usage. In our previous study [11] we introduced two tool categories: *static tools* whose main purpose is to present source code (*e.g.,* a source code browser); and *dynamic tools* whose main purpose is to present an application state (*e.g.,* a debugger). We annotated each session with time slots to know how much time participants spend using those tools (considering active windows, manual analysis).

We match those data with time slots participants spent answering questions. Considering a question occurrence, we compute a ratio number from an interval $[0, 1]$. It indicates the proportion of time spent answering the question using dynamic tools, *e.g.,* a number 0.3 denotes that a participant spent 30% of the time using dynamic tools and 70% of the time using static tools. The goal of this classification is to get us a first-order approximation of whether participants answered questions using static (navigating source code) or dynamic tools (runtime information, *e.g.,* instance variable values).

Categories. Figure 6 illustrates the tool usage by categories. For easy reading, we put static tool usage to the left, and dynamic tool usage to the right. Participants used 95% of the time static tools answering EFP (Expanding Focus Points) questions. Participants used dynamic tools more often answering FFP (Finding Focus Points, 48%), US (Understanding a Subgraph, 46%), and QGS (Questions over Groups of Subgraphs, 45%) questions.

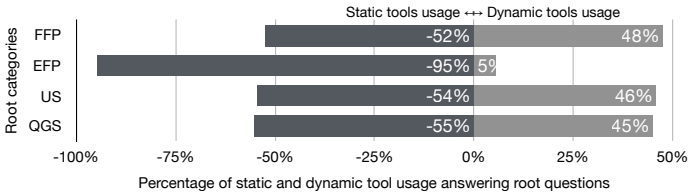

Fig. 6. Time spent answering root questions using static and dynamic tools.

Figure 7 offers a detailed look at the tool usage per unfamiliar and familiar codebase. Participants spent 27% more time using dynamic tools answering FFP questions on unfamiliar codebase sessions. Tool usage on EFP, US, and QGS questions was similar (varying by 6%, –8%, and 5% respectively) contrasting unfamiliar and familiar codebase sessions.

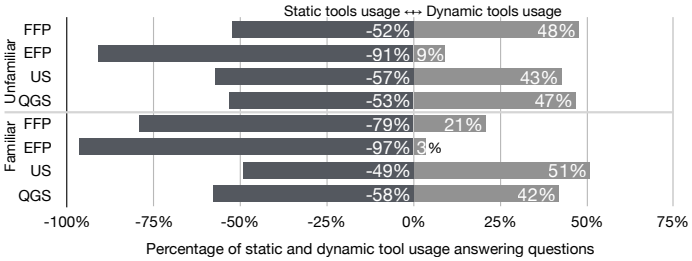

Fig. 7. Time spent answering root questions using static and dynamic tools.

Individual Questions. Table II includes tool usage information for each question. See second column *Tool usage by us*. We introduce five labels: *S* as strong static tool usage if 100% to 80% question occurrences were answered using static tools; *s* as static tool usage if 80% to 60% question occurrences were answered using static tools; *s/d* as both static and dynamic tool usage if 60% to 40% question occurrences were answered using static tools; *d* as dynamic tool usage if 40% to 20% question occurrences were answered using static tools; and *D* as strong dynamic tool usage if 20% to 0% question occurrences were answered using static tools.

Table II, column *Tool usage by Sillito*, includes programming tools and technique usage provided by Sillito *et al.* We

mark as *s* if Sillito *et al.* report a static tool usage to answer a question; *d* if they report dynamic tool usage; and *s/d* if they report both static and dynamic tool usage. Sillito *et al.* analyzed literature to provide the tool support and state-of-the-art techniques in answering catalogued questions. They did *not* analyze static and dynamic tool usage in their conducted sessions. We contrast both results in the rest of this section.

We observe that our participants used dynamic information in six questions, where Sillito *et al.* report only static tools support: *"(1) Which type represents this domain concept or this UI element or action?"* (FFP), *"(14) Where are instances of this class created?"* (EFP), *"(21) How are instances of these types created and assembled?"* (US), *"(25) What is the behavior that these types provide together and how is it distributed over the types?"* (US), *"(26) What is the 'correct' way to use or access this data structure?"* (US), and *"(41) How can we know that this object has been created and initialized correctly?"* (QGS). The dynamic tool usage not reported by Sillito *et al.* is particularly noticeable in the US category. Participants used dynamic tools in 82% (9 out of 11) US questions while Sillito *et al.* report 55% (6 out of 11) US questions, considering questions where we have tool usage information from our study and study by Sillito *et al.* We notice tendency using dynamic tool usage in our study and we discuss particular scenarios in the following Section VIII.

Observation 3. Participants often used dynamic tools answering FFP, US and QGS questions. EFP questions were answered predominantly using static tools.

VIII. Results: Tool Support Answering Questions

Our study provides insights about question complexity and static vs. dynamic tool usage answering those questions. To build an overall understanding of the pros and cons of liveness, we discuss each question category, contrasting our results with findings by Sillito *et al.*

Finding Focus Points (FFP). Sillito *et al.* report that four out of five questions are supported and answered using static analysis, text and lexical searches based on posible identifiers (names). We observe that participants used dynamic information too. To find out which classes correspond to UI widgets, question *"(1) Which type represents this domain concept or this UI element or action?"*, they inspected the widgets and then navigated to their classes. Participants also searched examples in dedicated example browsers, question *"(4) Is there a precedent or exemplar for this?"*. In particular, participant P4 (S8) went through the example list and explored their running instances (UI widgets).

Despite the fact that Sillito *et al.* report question *"(2) Where in the code is the text in this error message or UI element?"* with full support, human errors can significantly affect a task progress. P5 (S9) looked for a string that appeared on a dialog (question 2). He did *not* find it because of a spelling error and spent about forty minutes trying other techniques to find code relevant to his task. P5 could inspect the dialog and copy&paste the text, but he did not.

Expanding Focus Points (EFP). Sillito *et al.* found that participants used a debugger to check the accuracy of answers in this category. Participants set breakpoints to candidate locations and by running an application they revealed relevant classes and methods. Our participants moreover evaluated code snippets in debuggers and object inspectors to confirm the accuracy.

The Java language has a special syntax construct for constructors. For that reason Sillito *et al.* concludes that answering the question *"(14) Where are instances of this class created?"* is well supported by static analysis in Eclipse. The Pharo language does not have a special syntax construct for constructors, which are regular methods. Pharo developers have therefore two options to answer question number 14: (i) search for all class references, or (ii) put a breakpoint in an instance creation method and run the application. Our participants predominantly chose (ii), since they were usually interested in a specific case, not all cases.

Understanding a Subgraph (US). Sillito *et al.* report that 7 out of 13 questions in this category are supported by static analysis. Our participants also used dynamic tools to answer those 7 questions. They combined several activities: writing code snippets in playgrounds, inspectors, and debuggers; inspecting objects and graphical widgets; and reusing examples. In addition, the fact that Pharo opens a debugger whenever an unhandled exception appears, question *"(e8) What does the failure look like?"*, it encourages the runtime code exploration.

Participants in the study by Sillito *et al.* accessed values (objects) by putting breakpoints and executing applications. This is a limited way of obtaining runtime information, comparing it to our study. Due to the simple object inspection facilities, the increased number of occurrences of question *"(27) How does this data structure look at runtime?"* reflects the fact that participants tended to use dynamic information to answer the questions in this category. As a consequence, we notice that the increased opportunities to get dynamic information led to comprehend application by exploring it at runtime.

Questions over Groups of Subgraphs (QGS). Sillito *et al.* conclude that tool support answering questions in this category is not satisfactory and mainly supported by static analysis. We observe some situations where developers handled the difficulties by exploring applications at runtime. The question *"(34) How does the system behavior vary over these types or cases?"*, which is (according to Sillito *et al.*) difficult to answer especially when comparing behavior, participants P7 (S4) and P11 (S17) handled by writing different code snippets, executing the code and observing different behavior. Participants also altered the code of running applications to see an immediate feedback.

Questions *"(41) How can we know that this object has been created and initialized correctly?"*, *"(42) What will be (or has been) the direct impact of this change?"*, *"(43) What will be the total impact of this change?"*, and *"(44) Will this completely solve the problem or provide the enhancement?"* were also answered by observing applications at runtime in several situations. In the case of familiar sessions, participants

used and wrote test cases, question *"(e7) Do the test cases pass?"*, to keep a proper control of different application cases.

Summary. We observe that participants tend to use dynamic information (liveness) when possible, making some programming episodes easy to progress. US and QGS questions were predominantly answered using dynamic tools. However, the complexity of answering questions remains high, considering involved sub-questions.

US and QGS also involved many sub-questions about source code (EFP questions), causing transitions from one application (*e.g.,* inspector) to another (*e.g.,* source code browser). While Pharo dynamic tools are integrated, keeping navigation in one tool, we observe a disconnection when participants moved from dynamic information to static information (tool).

Observation 4. Participants accessed dynamic information frequently, making answering some questions easy.

IX. Threats to Validity

Empirical studies have to deal with tradeoffs between internal and external validity [39]. To deal with it properly, we decided on the following:

Internal validity. The transcript of each task activity was performed by the first author. When the author was unsure, all authors held discussions. To minimize biases in the interpretation, we built tools that allowed us to move between reports, transcripts, and audio and video recordings.

The identification of specific questions based on a user behavior may be inaccurate, as participants did not always verbalize their thoughts explicitly. The subsequent synthesis of general questions also suffer from uncertainty. For example, the questions *"(13) When during the execution is this method called?"* and *"(31) Which execution path is being taken in this case?"* can be difficult to distinguish. To minimize the inaccuracy, we contrasted different scenarios and tool usage answering the same questions.

External validity. The session tasks may not be representative for the Pharo IDE, and thus: (i) may not reveal significant benefits of the Pharo tools, (ii) may produce different results on the number of question occurrences, question complexity, and tool usage. We only define tasks on unfamiliar codebase and we left the task on familiar codebase to the participants.

We had a limited choice of participants who may share similar development techniques. Our participants were all males and studies like the two of Beckwith *et al.* have shown differences between males and females while tinkering [40], however, the relationships is not obvious at all [41].

Generalizing results is restricted, due to the difficulty to conduct such studies on a large scale: transcribing one session is time-consuming (about 2 days per session). Although the number of sessions is similar to related work (Fritz *et al.* [34], 11 interviews; LaToza *et al.* [33], 13 sessions; Sillito *et al.* [1], 27 sessions), due to: the limited amount of sessions and participants; the lack of diversity in participants, tools and tasks; and session durations, it is not certain the results will generalize.

X. Summary

Additional questions. We document eight additional developer questions. Five questions are related to the fact that the Pharo language is dynamically typed and navigating in source code is not always straightforward. Similarly, method return types, method argument types, and variable types are not explicit and thus require extra investigation.

Two new questions are related to using and writing test cases. The most frequent new question is *"(e7) Do the test cases pass?"* and was asked predominantly in sessions on familiar codebases. Similarly to the observation by Sillito *et al.* using test cases was uncommon. Test cases appeared in four out of seventeen sessions.

Question *"(e8) What does the failure look like?"*, is related to the fact that Pharo opens a debugger whenever an unhandled exception appears. Participants were thus naturally encouraged to observe issues and the corresponding runtime information.

Question occurrences and tool usage. Our participants asked more frequently questions that involve runtime information to answer them. In particular, question *"(27) How does this data structure look at runtime?"* This is due to the ease of acquiring this information in Pharo. We also observe scenarios where participants gained knowledge or confidence about source code exploring applications at runtime, even in situations where Sillito *et al.* report static analysis as sufficient. To access the runtime information they used existing examples, wrote code snippets, inspected graphical widgets, and manipulated objects in inspector and debugger windows.

Question complexity. Despide having observed benefits using dynamic information, complex questions still remained, involving many sub-questions. Having dynamic information available is thus not a panacea for programming tasks and further research is necessary to find out how liveness feedback can significantly improve developer performance.

XI. Conclusion

Documenting developer information needs is an important research task that is regularly investigated in different contexts. We partially replicated the study by Sillito *et al.* and found 1,161 developer questions in 17 programming sessions covering about 13 hours of video. The sessions were performed in Pharo, a Live Programming environment.

Participants in the study by Sillito *et al.* favored static observations, while our participants favored dynamic information, as it was easily accessible in many different ways (not only with a debugger, see Section II-C for details). However, many program explorations involved many sub-questions, dealing with static and dynamic informations at once. Thus, more research needs to be done to improve this situation and make the static and dynamic information seamlessly inter-connected. Studying such new development environments might change what questions developers ask and how they answer them.

Acknowledgments We thank Renato Cerro for editing. Kubelka is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional/2013-63130188. Bergel thanks Lam Research for sponsoring part of this effort.

REFERENCES

[1] J. Sillito, G. Murphy, and K. De Volder, "Asking and Answering Questions during a Programming Change Task," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 434–451, July 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.26

[2] T. D. LaToza and B. A. Myers, "Developers Ask Reachability Questions," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806829

[3] A. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, May 2007, pp. 344–353. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.45

[4] E. Duala-Ekoko and M. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 266–276. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227187

[5] J. Sillito, "Asking and answering questions during a programming change task," Ph.D. dissertation, University of British Columbia, Feb 2006. [Online]. Available: https://open.library.ubc.ca/cIRcle/collections/831/items/1.0052042

[6] N. Juristo and S. Vegas, "The role of non-exact replications in software engineering experiments," *Empirical Software Engineering*, vol. 16, no. 3, pp. 295–324, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10664-010-9141-9

[7] S. L. Tanimoto, "A Perspective on the Evolution of Live Programming," in *Proceedings of the 1st International Workshop on Live Programming*, ser. LIVE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 31–34. [Online]. Available: http://dl.acm.org/citation.cfm?id=2662726.2662735

[8] C. Parnin, "A History of Live Programming," 01 2013. [Online]. Available: http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/

[9] E. Sandewall, "Programming in an Interactive Environment: The "Lisp" Experience," *ACM Comput. Surv.*, vol. 10, no. 1, pp. 35–71, Mar. 1978. [Online]. Available: http://doi.acm.org/10.1145/356715.356719

[10] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*. Reading, Mass.: Addison Wesley, May 1983. [Online]. Available: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[11] A. Author(s), "Anonymous Title for the Double Blind Review," in *Anonymous Proceedings*, ser. Anonymous '18. Anonymous Address: ACM, 2018, pp. 1–12.

[12] P. Rein, S. Lehmann, T. Mattis, and R. Hirschfeld, "How Live are Live Programming Systems?" *Proceedings of the Programming Experience 2016 (PX/16) Workshop on - PX/16*, 2016. [Online]. Available: http://dx.doi.org/10.1145/2984380.2984381

[13] D. Ungar and R. Smith, "SELF: the power of simplicity (object-oriented language)," *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*, 1988. [Online]. Available: http://dx.doi.org/10.1109/CMPCON.1988.4851

[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future," *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97*, 1997. [Online]. Available: http://dx.doi.org/10.1145/263698.263754

[15] J. Noble and S. McDirmid, Eds., *Proceedings of the 2nd International Workshop on Live Programming, LIVE 2016, Rome, Italy, July 17, 2016*, 2016.

[16] D. Ogborn, G. Wakefield, C. Baade, K. Sicchio, and T. Goncalves, Eds., *Proceedings of the Second International Conference on Live Coding*, 2016.

[17] S. L. Tanimoto, "VIVA: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, 1990.

[18] M. M. Burnett, J. W. Atwood, and Z. T. Welch, "Implementing level 4 liveness in declarative visual programming languages," in *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE, 1998, pp. 126–133.

[19] S. McDirmid, "Living it up with a live programming language," in *ACM SIGPLAN Notices*, vol. 42. ACM, 2007, pp. 623–638.

[20] J. Edwards, "Subtext: uncovering the simplicity of programming," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 505–518, 2005.

[21] S. McDirmid and J. Edwards, "Programming with managed time," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014, pp. 1–10.

[22] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in UI programming," in *ACM SIGPLAN Notices*, vol. 48. ACM, 2013, pp. 95–104.

[23] R. DeLine, D. Fisher, B. Chandramouli, J. Goldstein, M. Barnett, J. F. Terwilliger, and J. Wernsing, "Tempe: Live scripting for live data." in *VL/HCC*, 2015, pp. 137–141.

[24] G. C. D. Team, "Google Chrome Development Tools," 2017. [Online]. Available: https://developers.google.com/web/tools/chrome-devtools/

[25] R. D. Team, "React: A Javascript Library For Building User Interfaces," 2017. [Online]. Available: https://facebook.github.io/react/

[26] Oracle, "Java Platform Debugger Architecture," 2014. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/

[27] K. Eliott, "Swift Playgrounds—Interactive Awesomeness," 2016. [Online]. Available: http://bit.ly/Swift-Playgrounds

[28] S. Hanselman, "Interactive Coding with C# and F# REPLS," 2016. [Online]. Available: http://bit.ly/InteractiveCodingCF

[29] K. Uhlenhuth, "Introducing the Microsoft Visual Studio C# REPL," Nov 2015. [Online]. Available: https://channel9.msdn.com/Events/Visual-Studio/Connect-event-2015/103

[30] C. Connect, "Code Connect is joining Microsoft," 2016. [Online]. Available: http://comealive.io/Code-Connect-Joins-Microsoft/

[31] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009. [Online]. Available: http://pharobyexample.org

[32] A. Chiş, O. Nierstrasz, A. Syrel, and T. Gîrba, "The Moldable Inspector," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 44–60. [Online]. Available: http://doi.acm.org/10.1145/2814228.2814234

[33] T. D. LaToza and B. A. Myers, "Hard-to-answer Questions About Code," in *Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/1937117.1937125

[34] T. Fritz and G. C. Murphy, "Using Information Fragments to Answer the Questions Developers Ask," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806828

[35] B. de Alwis and G. C. Murphy, "Answering Conceptual Queries with Ferret," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368092

[36] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 492–501. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134355

[37] A. Author(s), "Anonymous Title for the Double Blind Review," in *Anonymous Proceedings*, ser. Anonymous '14. Anonymous Address: ACM, 2014, pp. 1–11.

[38] ——. (2019, Jan) Anonymous Dataset for the Double Blind Review. [Online]. Available: https://www.dropbox.com/sh/swgnwg75z51hr2w/AACeT3QiixvrVZv0wc4TJ1HFa?dl=0

[39] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 9–19.

[40] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, "Effectiveness of end-user debugging software features: Are there gender issues?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2005, pp. 869–878.

[41] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006, pp. 231–240.