Asking and Answering Questions during a Programming Change Task in Pharo Language

Juraj Kubelka Alexandre Bergel Romain Robbes

PLEIAD Laboratory, Department of Computer Science (DCC)
University of Chile, Santiago, Chile
{jkubelka,abergel,rrobbes}@dcc.uchile.cl

Abstract

Previous studies focus on the specific questions software engineers ask when evolving a codebase. Though these studies observe developers using statically typed languages, little is known about the developer questions using dynamically typed languages. Dynamically typed languages present new challenges to understanding and navigating in a codebase and could affect results reported by previous studies.

This paper replicates a previous study and presents the analysis of six programming sessions made in Pharo, a dynamically typed language. We found a similar result when comparing sessions on an unfamiliar codebase with the previous work. Our result on the familiar code greatly deviates from the replicated study, likely caused by different tasks and development strategies. Both missing type information and test driven development affected participant behavior and prudence on codebase understanding, where some participants made changes based on assumptions.

We provide a set of questions that are useful in characterizing activity related to the use of a dynamically typed language and test-driven development — questions not explicitly considered in previous research. We also present a number of issues that we would like to discuss during the PLATEAU workshop.

1. Introduction

Programming environments have tremendously improved over the last decade. What were previously simple text editors are now fully fledged studios for code production. Navigating between source code elements is now supported in many different ways by most programming environments.

Sillito *et al.* [9] (herein designated as *Sillito*) made a number of observations on developer navigation. They identify four question categories and levels of tool support for getting answers. They conducted two studies observing software programmers of statically typed languages C++, C, C#, and Java. In their first study, the participants worked on a change task for one unique open source project, ArgoUML[1], of which they were not familiar. The second study was conducted in an industrial setting including software engineers working on a change task of familiar codebase. The context setting used by Sillito in their experiment does not cover some commonly found software engineering practices. For example, they only consider statically typed languages, one industrial codebase, and one open source codebase.

Research question. Our work replicates the experiment by Sillito *et al.* and validates it in a new scenario. The participants worked on tasks in Pharo, a dynamically typed programming language, and in distinct open source software systems. The dynamically typed languages present new challenges to understanding and navigating in a codebase. Both aspects — dynamically typed language and different codebases could affect results reported by Sillito. In summary, our research question is:

> *Are findings presented by Sillito applicable to programming change tasks using the Pharo programming language?*

Pharo. The Pharo[2] environment (Pharo IDE) illustrated in Figure 1 is largely different from the ones considered in the Sillito experiment. The Pharo programming environment offers a set of expressive and flexible programming tools. The System Browser (2) is the main tool for writing and reading source code. Navigation within the source code is essentially based on the SendersOf (4), ImplementorOf, and UsersOf tools; whenever a user asks to where a particular method is called, or asks for method definition, field reference, or class

PLATEAU '2014, October 21, 2014, Portland, Oregon, United States.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] `http://argouml.tigris.org`, verified September 2014

[2] `http://pharo.org`, verified September 2014

reference, a new window appears with a corresponding list. In addition, test driven development is supported by the Test Runner (1). Incremental and live programming is supported by the Workspace (6) and Transcript (7), a kind of Unix-like terminal for Pharo.

Pharo language is a dialect of the Smalltalk language [2]. It is a pure object oriented and dynamically typed language.

Results. We observe the following:

- The number of question occurrences on the unfamiliar code has similar results compared with the study number one by Sillito.
- The study on the familiar code indicates great deviation, likely caused by different tasks and development strategies: one participant did not use test cases and was considerably cautious; another participant relied heavily on the accuracy of tests and did not spend much time reading the codebase.
- Both missing type information and Test Driven Development (TDD) affected a participant's judgement on codebase understanding. Some participants made changes based on assumptions: if a test scenario (a test case or manual testing) worked, they went further with a particular task; if a test scenario did not work, they carefully studied a particular implementation in the codebase.

Finally we provide a set of questions that are useful for characterizing activity related to the use of a dynamically typed language and test-driven development — questions not explicitly considered by Sillito.

Open questions. Our preliminary work makes a number of assumptions that we hope to verify during the PLATEAU workshop; we provide the questions we would like to raise during the workshop.

Outline. Section 2 describes settings of our study. Section 3 describes the analysis process on the collected data. Section 4 compares our results against the study by Sillito, reports our observations, and aggregates new questions. Section 5 considers validity to our study. Section 6 concludes our work, and discusses related and future work. Section 7 raises open issues for the PLATEAU workshop.

2. Programming Study

This section describes settings of our exploratory study.

2.1 Initial Considerations for Experimental Design

The intention of the experiment is to conduct *partial replication* [3]. Here we describe the setting differences comparing them to the study by Sillito.

Pharo language. We use dynamically typed Pharo language. Participants in the studies by Sillito use statically typed languages, *e.g.,* Java, C++, or C#. We expect that participants will be interested in variable types and method return types.

Kleinschmager *et al.* observe in their empirical study that static type systems improve maintainability of software systems [4]. This is an aspect which may affect our partial replication. Pharo however supports live-programming that enables manipulation and inspection of data and program execution, though it may diminish the observation by Kleinschmager *et al.*

Different IDE. Each IDE comes with a different set of tools and hence may affect developer work. To our knowledge there is no official comparison of Pharo with the tools used in the sessions by Sillito. We therefore do not know if using Pharo IDE affects our study.

Development process. Pharo developers commonly use Test Driven Development (TDD). Oram *et al.* conduct a systematic review of TDD research [8], and are interested in knowing if there is evidence of TDD improving product quality and productivity. They conclude that it varies and there is no common agreement about TDD benefits. Sillito does not discuss what development technique the participants use. Nevertheless the development process could affect our study.

Pair programming for unfamiliar codebase. Sillito's participants worked in pairs on each task. The discussion between developers helped them understand what information they were looking for. We choose one-man programming sessions because this is the way our participants usually work. In our case the observer "shadows" the position and a particular developer explains his/her actions to the observer. We do not expect the one-man programming to affect our study.

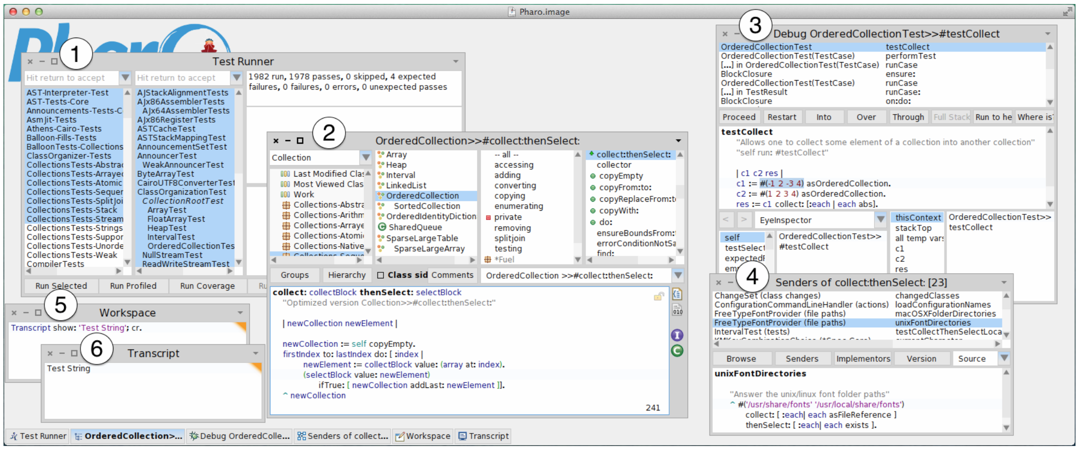
Task definition for unfamiliar codebase. Sillito selects specific issues from ArgoUML issue tracker. They are without any reference to the codebase, *e.g.,* classes or methods. Our assignments contain a snippet of an example related to a particular task; it is a common practice to report an issue for the used codebase. We expect that our participants ask less questions related to finding focus points in the codebase.

Task definition for familiar codebase. Sillito conducts the tasks on the familiar codebase in one company. Our participants — although working in companies — are better described as individual programmers; they usually work alone and comunicate with others through internet. As the participants worked on their codebase, we do not expect individual programming to influence our study.

Data collection for familiar codebase. Sillito uses an audio recording and takes notes during each session. We use audio and screen recording. Our technique may improve the accuracy of the analysis, but we do not expect the recording system to affect our study.

2.2 Data Collection

We used three data collection techniques: the think-aloud protocol, screen captured videos, and interviews. In the think-aloud protocol, we asked participants to verbalize their



```
1982 run, 1978 passes, 0 skipped, 4 expected
failures, 0 failures, 0 errors, 0 unexpected passes
```

```
collect: collectBlock thenSelect: selectBlock
   "Optimized version Collection>>#collect:thenSelect:"

   | newCollection newElement |

   newCollection := self copyEmpty.
   firstIndex to: lastIndex do: [ :index |
         newElement := collectBlock value: (array at: index).
         (selectBlock value: newElement)
            ifTrue: [ newCollection addLast: newElement ]].
   ^ newCollection
```

```
testCollect
   "Allows one to collect some element of a collection into another collection"
   "self run: #testCollect"

   | c1 c2 res |
   c1 := #(-1 2 -3 4) asOrderedCollection.
   c2 := #(1 2 3 4) asOrderedCollection.
   res := c1 collect: [:each | each abs].
```

```
ChangeSet (class changes)                 changedClasses
ConfigurationCommandLineHandler (actions) loadConfigurationNames
FreeTypeFontProvider (file paths)         macOSXFolderDirectories
FreeTypeFontProvider (file paths)         unixFontDirectories
IntervalTest (tests)                      testCollectThenSelectLoca
```

```
unixFontDirectories

   "Answer the unix/linux font folder paths"
   ^ #('/usr/share/fonts' '/usr/local/share/fonts')
      collect: [ :each| each asFileReference ]
      thenSelect: [ :each| each exists ].
```

```
Transcript show: 'Test String'; cr.
```

```
Test String
```

Figure 1. Standard Pharo Programming Environment with (1) Test Runner — for test driven development, (2) System Browser — a main tool for writing and reading source code, (3) Debugger, (4) SendersOf/ImplementorsOf/UsersOf — for navigating method and type references and declarations, (5) Workspace and (6) Transcript — for incremental and live programming.

thoughts while solving a given task. It permits us to identify the developer questions.

After each session we also conducted semi-structured interviews in which the participants were asked to comment on the challenges experienced during the sessions and divergencies in their common day-to-day work. The interviews lasted about 5 minutes.

The screen and audio recording, and interviews were captured using QuickTime Player on the Apple OS X operating system, and Camtasia on the Microsoft Windows operating system. The study contains 6 different programming sessions and about 5 hours of videos.

2.3 Participants

We recruited six participants consisting of doctoral students, and staff from the department of Computer Science at the University of Chile, and professional developers from distinct small local companies. As the community located in Santiago is relatively small and the programmers know each other, participants were invited to the study either via email or through personal encounters. We restrict our experiment to Pharo programmers.

The software development experience among the participants ranged from 5.5 to 22 years, with a median of 11 years. The experience in a Smalltalk programming language ranges from 0.5 to 16 years, with a median of 7 years. The participants consisted of 1 Ph.D. student, 2 professors, and 3 professional developers. Details are illustrated in Table 1.

2.4 Tasks

We observed six programming sessions (one session per participant). Table 1 illustrates how the sessions were distributed. Being familiar or not with a codebase when carrying out a programming activity is a significant factor, on which our experiment is articulated: 3 sessions were conducted using a *familiar codebase*, 1 session on a *partially familiar codebase*, and 2 sessions on an *unfamiliar codebase*. The participants working on the familiar code chose their tasks in advance. The participants working on partially familiar or unfamiliar codebase worked on tasks we prepared for them.

All sessions were conducted on the participant devices and were scheduled on average with two-days notice.

2.4.1 Familiar Codebase

We qualify a participant P as familiar to a codebase if P is one of the authors of the codebase.

We asked each participant to choose a programming task of his/her project on which he/she can work about 40 minutes. We stressed that it was not important to finish the task at the end of the session. Each participant chose a task and prepared his/her device in advance.

2.4.2 Partially Familiar and Unfamiliar Codebase

We qualify a participant P as partially familiar with a codebase if P has previous experience with the codebase — for example, the participant used it before.

We define unfamiliar codebase in which a participant P has little to no knowledge of where the minimal knowledge is not beneficial to completing the task.

We decided to differentiate partially familiar and unfamiliar codebase knowledge when participant P3 reported that knowledge of other parts of the framework helped him be more oriented on the focused code.

We asked each participant to work on a particular task. All tasks were made on Roassal2[3] in Pharo. Roassal2 is a visualization engine that follows the common model-view-controller pattern. Each participant worked on a task that was different from that of other participants. The purpose was to

[3] `http://objectprofile.com/Roassal.html`, verified September 2014

indicate which code structure could highlight problems in order to better understand it.

Each assignment included a prepared Pharo image with a snippet of an example related to a particular task and task explication. Participants copied their assignment to their device and after basic checks of the recording software and code, started to work.

2.5 Study Setting

Participants working on partially familiar and unfamiliar codebase completed the study using the Pharo (version 3). We choose the Pharo since we have substantial experience developing in it and we know the characteristics of the language and IDE. This increases our confidence in our interpretation of the participant actions.

Participants working on the familiar codebase completed the study using the Moose platform (version 5). It is based on Pharo version 3 and it includes additional tools. It was not our choice, but the participants' preference. Participant P6 even demanded to work in Moose 5, because he uses some frameworks included in that image. The session settings are illustrated in the Table 1.

Our purpose is not to compare a developer's behavior in Moose and in Pharo. Nevertheless we did not see a significant change in participant behavior using Moose. Only participant P5 used the Moose Inspector tool regularly in order to understand object compositions. A similar function is available in Pharo as Tree Inspector tool.

The participants were permitted to use any of the features of the IDEs and any of the documentation resources. They were advised to proceed with their work as usual.

The programming studies were conducted individually on the university premises. Sessions S2, S4, and S6 were conducted in participant offices. The sessions S1 and S3 were conducted at other locations available to the university staff. Session S5 was done at the participant's private premise without an observer. Before this session S5, we conducted another session with participant P5, but the screencast was lost because of technical problems. Participant P5 was willing to run another session in his office a few days later. He built it on the work of the original session. As he was experienced from the first session, we do not detect any threats to validity.

Before each study, we setup the participant device and explained the procedure of the session, in particular, how the think-aloud protocol works. Other than this explanation, we have not done any training on the think-aloud protocol.

Once the participant was satisfied and his device was ready, we started a screen and voice recording software and DFlow framework [7] for tracking additional information about developer behavior.

3. Data Analysis

This section describes the analysis process on the collected data.

Our analysis focuses on the questions the participants ask when developing. Our goal is to identify those questions and compare them with the findings of Sillito [9]. Our method of analysis involves two techniques: (i) identifying the concrete questions, (ii) generalizing of concrete questions and mapping to existing questions from Sillito.

Identification of concrete questions. In this phase, we go through the recorded videos and produce the semi-structured transcript, presented below ("he" refers to the participant):

- 06:35-08:08 he asks Q*"How is the background created for the parent menu?"*
  - 06:39-06:44 he goes to method `createParentMenu: background:` observing the implementation where he sees another method Q*"What does the method look like?"* which creates background
  - 06:44-07:05 then he asks Q*"Why does it not do the same things [parent menu label and background color] at the same time?"*

We verbalized each action as a concrete question annotated by `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 figured out from the actions, *e.g.,* P1 jumped from the code where `TRMouseClick` class was used and observed its class definition and its method names. This action is phrased into the question *"What are the parts of TRMouseClick?"*

Generalization of concrete questions. After identifying specific questions, we then synthesized generic questions that abstract the specifics of a given task. We include generic questions in the transcript annotated by `GQ` symbol, *e.g.,* GQ*"(23) How is this feature or concern (object ownership, UI control, etc.) implemented?"* The transcript below is the final version of the above transcript:

Session	Participant	Experience [years]	Smalltalker [years]	Position	Task	Knowledge	Tools
S1	P1	5.5	3	Ph.D. Student	Enhancement	Unfamiliar code	Pharo
S2	P2	15	11	Professor	Enhancement	Unfamiliar code	Pharo
S3	P3	5	1.5	Professional	Enhancement	Partial knowledge	Pharo
S4	P4	20	13	Profesional	Enhancement	Own code	Moose
S5	P5	7	0.5	Profesional	Enhancement	Own code	Moose
S6	P6	22	16	Profesor	Enhancement	Own code	Moose

Table 1. Participant information for each session: session number, participant number, programming experience as developer in any language, programming experience in Smalltalk language, current position, task, codebase knowledge, and tools used.

- 06:35-08:08 he 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 he goes to method `createParentMenu: background:` observing the implementation where he 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 he 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?”*

During this process we used the list of questions identified by Sillito [9]. For instance, during our study a participant raised the question *“How is the background created for the parent menu?”*, and we map it to the question *“(23) How is this feature or concern (object ownership, UI control, etc.) implemented?”* proposed by Sillito.

Some concrete questions are not conveniently translatable to the list of questions presented by Sillito, *e.g.,* Q*“Why does the test case fail?”* In such cases questions presented by Ko *et al.* [5] are used, *e.g., “(r1) What does the failure look like?”*

If none of the questions proposed by Sillito and Ko are related to a question of our study, we abstract the question; for example we map the question *“Is the `R3CubeShapeclass` tested?”* to the generic question *“(e6) Is this entity tested?”* Each new question is added to one of question categories identified by Sillito.

Results. Table 2 gives the frequency of each question during the sessions. The first column lists general questions per category in the same way as illustrated by Sillito. Questions from Sillito begin with a number, *e.g., “6. What are the parts of this type?”* Our new questions begin with *e* followed by a number in brackets, *e.g., “(e3) Which abstract methods should be implemented to this type?”* The question *“(r1) What does the failure look like?”* is taken from the study by Ko *et al.* We use only this question from Ko *et al.* because their questions are more generally formulated and we are able to find corresponding questions in the work by Sillito. Even question (r1) could be mapped to one of the Sillito’s questions, but in particular cases we are not able to formulate participant needs more precisely.

The columns are grouped to the sessions on the unfamiliar codebase and to the sessions on the familiar codebase. Columns S1-S6 are particular sessions of this study. Columns #1 and #2 are the sum of particular sessions, and SM1 and SM2 correspond to Sillito’s results.

4. Results

This section is structured as following: (i) Section 4.1 compares results on the unfamiliar and familiar codebase. Section 4.2 debates aspects of dynamically type language on answering questions. Section 4.3 discusses aggregated questions.

4.1 Session Comparison

4.1.1 Results on Unfamiliar Codebase

The first graph in Figure 2 gives the distribution of the question occurrences per category and session. Each bar represents the percentage of the number of questions in a particular session. The bar *#1* is the number of question occurrences in the sessions on the unfamiliar codebase. The bar *SM1* corresponds to the results from the study number one by Sillito.

Finding focus points. Deviations in this category between each conducted session may be partially explained by different knowledge about the Roassal 2 framework. P2 never used the API. P1 used a similar project before (Roassal in version 1) and was never interested in the internal implementation. Previously, P3 worked on some portion of another part of the framework. It helped him become more oriented to the framework.

Expanding focus points. Deviations in this category between each conducted session may be partially explained by different strategies and amount of code each participant observed. Participant P1 was highly interested in understanding a particular behavior and was navigating repeatedly over several method calls made on several objects. P2 navigated

	Sessions on Unfamiliar Code					Sessions on Familiar Code				
Question Types per Category	S1	S2	S3	#1	SM1	S4	S5	S6	#2	SM2
Finding Focus Points										
1. Which type represents this domain concept or this UI element or action?			1	1	8		2	4	6	
2. Where in the code is the text in this error message or UI element?					4					
3. Where is there any code involved in the implementation of this behavior?	3	11	2	16	10					2
4. Is there a precedent or exemplar for this?	1	1	1	3	4		2		2	4
5. Is there an entity named something like this in that unit (project, package, or class, say)?	4		1	5	11		1	1	2	1
Total in the category	8	12	5	25	37		5	5	10	7
Expanding Focus Points										
Types and Static Structure										
6. What are the parts of this type?	9	2	1	12	11	1	4		5	1
7. Which types is this type a part of?			1	1	2		1		1	
8. Where does this type fit in the type hierarchy?	1			1	10					
9. Does this type have any siblings in the type hierarchy?					2	1			1	
10. Where is this field declared in the type hierarchy?		1		1	2	2			2	
11. Who implements this interface or these abstract methods?	2	2	4	8	5					
Extra Questions on Types and Static Structure										
(e1) Where is the method defined in the type hierarchy?	1			1						
(e2) What are the correct argument names of this method?						4			4	
(e3) Which abstract methods should be implemented to this type?								2	2	
Incoming Connections										
12. Where is this method called or type referenced?	7	4	1	12	33	1	1	6	8	2
13. When during the execution is this method called?					4					1
14. Where are instances of this class created?					8					
15. Where is this variable or data structure being accessed?	3	2	3	8	8					3
16. What data can we access from this object?					1					1
Outgoing Connections										
17. What does the declaration or definition of this look like?	5	12	15	32	13	21	1	19	41	5
18. What are the arguments to this function?	3		1	4	10					
19. What are the values of these arguments at runtime?			6	6	4					1
20. What data is being modified in this code?			2	2	2			1	1	
Extra Questions on Outgoing Connections										
(e4) What method implementation corresponds to my question?	1			1						
(e5) What is the variable's type or what is the method's return type?	1	4	2	7		4	1	2	7	
Total in the category	33	27	36	96	115	34	8	30	72	14
Understanding a Subgraph										
Behavior										
21. How are instances of these types created and assembled?	2	1	2	5	9					
22. How are these types or objects related? (whole-part)	1		1	2	3		1		1	
23. How is this feature or concern (object ownership, UI control, etc.) implemented?	6		7	13	12		2	1	3	3
24. What in this structure distinguishes these cases?					2					1
25. What is the behavior that these types provide together and how is it distributed over the types?	5	1	7	13	7			1	1	1
26. What is the "correct" way to use or access this data structure?					3		4		4	3
27. How does this data structure look at runtime?	2	3	4	9	3		9		9	3
Data and Control Flow										
28. How can data be passed to (or accessed at) this point in the code?		5		5	4					1
29. How is control getting (from here to) here?					2					
30. Why is control not reaching this point in the code?		1		1	6			1	1	2
31. Which execution path is being taken in this case?			1	1	7					2
32. Under what circumstances is this method called or an exception thrown?			2	2	8					
33. What parts of this data structure are accessed in this code?					3					
Extra Questions on Data and Control Flow										
(r1) What does the failure look like?		1	4	5		14	3	1	18	
Total in the category	16	12	28	56	69	14	19	4	37	16
Questions over Groups of Subgraphs										
Comparing or Contrasting Groups										
34. How does the system behavior vary over these types or cases?	2	1	2	5	2	2	2		4	1
35. What are the differences between these files or types?					1	2	1	1	4	8
36. What is the difference between these similar parts of the code (e.g., between sets of methods)?			1	1	3		1		1	4
37. What is the mapping between these UI types and these model types?					4					
Change Impact										
38. Where should this branch be inserted or how should this case be handled?			8	8	5		4	1	5	2
39. Where in the UI should this functionality be added?					4					2
40. To move this feature into this code, what else needs to be moved?										2
41. How can we know that this object has been created and initialized correctly?			1	1	2					
42. What will be (or has been) the direct impact of this change?		6	4	10	7	3	2	1	6	15
43. What will the total impact of this change be?			1	1						9
44. Will this completely solve the problem or provide the enhancement?		4	7	11	3		4		4	2
Extra Questions on Change Impact										
(e6) Is this entity or feature tested?						1			1	
(e7) Do the test cases pass?						18			18	
Total in the category	2	11	24	37	31	26	14	3	43	45
Total	59	62	93	214	252	74	46	42	162	82

Table 2. The number of question occurrences of each type that were asked in each session. Questions from Sillito begin with number, *e.g., "6. . . . "*, our new questions begin with *e*, *e.g., "(e3) . . . "*, question *(r1)* comes from the study by Ko *et al.* Columns S1-S6 are question occurrences for each session. #1 is the sum of occurrences on the unfamiliar codebase. #2 is the sum of occurrences on the familiar codebase. SM1 and SM2 are results from the study by Sillito.

within a significantly smaller set of entities. P3 was not navigating much and made a lot of assumptions about the codebase. The majority of the assumptions were right.

Understanding a subgraph. Deviations in this category between each conducted session may be partially explained by different amounts of code each participant observed and difficulties in understanding it. P1 spent a significant amount of

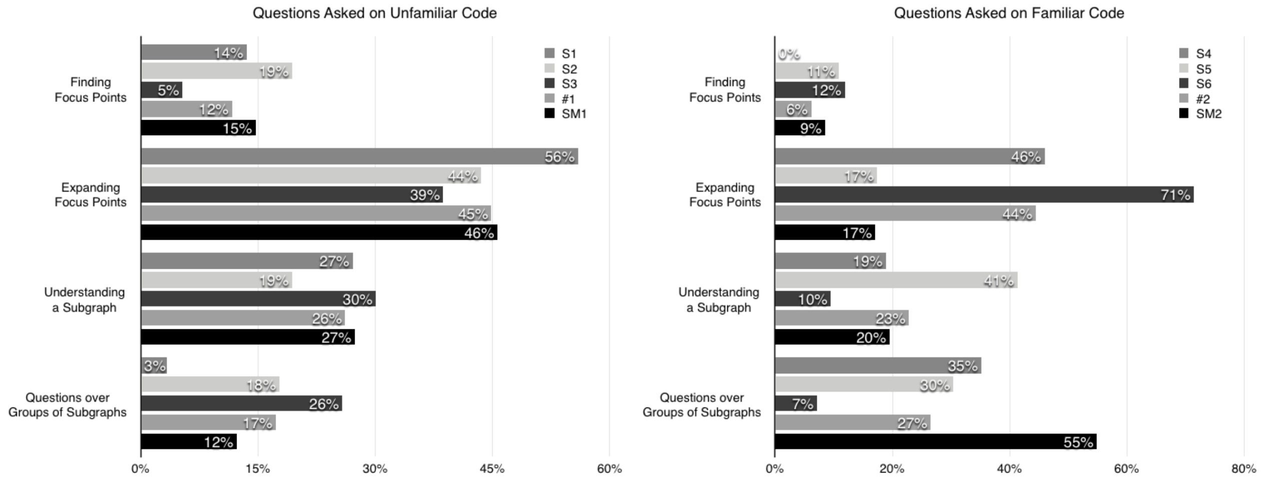


Figure 2. Questions asked on the unfamiliar and familiar code per category and session. Each bar represents the percentage of the number of questions in a particular session (S1-S6). The bars *#1* and *#2* are the numbers of question occurrences in the sessions on the unfamiliar and familiar codebase. The bars *SM1* and *SM2* correspond to the results from the study by Sillito.

time on fully understanding subgraphs which did not lead to the solution. P2 understood relatively small and isolated subgraph of the framework. P3 did not understand a portion of codebase that extensively uses the language reflection API; even observation at runtime was not helpful because the values of interested variables were symbols (specialized string values). He tried different strategies to understand the subgraph.

Questions over groups of subgraphs. Deviations in this category between each conducted session may be partially explained by different achievements and strategies. P1 did not finish the task and made a relatively small amount of changes. It implies that he did not spend much time by contrasting the change impact or by contrasting different behaviors. P2 regularly made changes to the codebase and manually tested the impact or compared actual and original version of the codebase. P3 made incremental changes which he tested and considered how to handle failures in particular cases.

Overall question occurrences. Distribution of total number of question occurrences in groups (#1) is similar to the study by Sillito (SM1). A notable difference is in the fourth category caused mainly by session S3. In this session, P3 made the most significant progress in comparison with session S1 and S2.

4.1.2 Results on Familiar Codebase

The second graph in Figure 2 gives distribution of the number of question occurrences per each category and session. Each bar represents the percentage of the number of questions in a particular session. The bar *#2* illustrates the number of question occurrences in the sessions on the familiar codebase. The bar *SM2* depicts corresponding results from the study number two by Sillito.

All the sessions were made on the codebase that the participants own and regularly work on it. Prior to the session, each participant was asked to select a task. All the participants worked on codebase enhancement.

Finding focus points. Deviations in this category between each conducted session may be partially explained by complexity of session tasks. P4 knew exactly where to start, what to change, and how to change it. Participants P5 and P6 spent some time in locating some entities in their codebase or in other APIs they use. For example P5 wanted to use a tree structure and was interested if there is any support in Pharo.

Expanding focus points. Deviations in this category between each conducted session may be partially explained by different tasks and development strategies. The codebase used in the session S4 is driven by test cases. When P4 reached the expected changes, he regularly executed test cases, fixing one after another and expanding focus points from a method in a debugger tool to various parts of codebase for which he needed to understand. The codebase used in session S5 has extensive class hierarchies and P5 occasionally clarified the definition of used classes. P6 regularly used two code browsers; one for navigating, one for editing. This could be the reason why he repeatedly navigated the same part of codebase. The part of the codebase he was changing is not covered by test cases. It could also be the reason for his caution. He preferred to double check his work if there were any discrepancies.

Understanding a subgraph. Deviations in this category between each conducted session may be partially explained by different tasks and development strategies. P4 used test driven development and when he reached the enhancement, he spent half of the session fixing dependent codebase by iterating between running test cases and fixing code. P5 regularly observed complex XML structures at runtime. P6 altered a relatively small subgraph of his codebase and the behavior was clear to him.

Questions over groups of subgraphs. Deviations in this category between each conducted session may be partially explained by the use of different tasks and development strategies. Participants in the sessions S4 and S5 regularly executed test cases contrasting the change impact. P5 got stuck thinking about how to handle a particular case. P6 worked on relatively isolated enhancement which he joined to the rest of the codebase and conducted manual testing without significant problems.

Overall question occurrences. Deviations between total number of question occurrences in groups (#2) and the study number two by Sillito (SM2) are significant in the second group (Expanding Focus Point) and fourth group (Questions over Groups of Subgraphs): 71% versus 44%, and 27% versus 55%. These differences are partially explained by the use of different tasks or development strategies. Development strategies of sessions S4 and S6 differ significantly. The test driven development is used for all changes made in session S4. On the contrary no test case was used in session S6. In session S5 it was important to understand data structure at runtime while participants P4 and P6 did not need to understand the data structures in such detail.

4.2 Pharo-Specific Aspects on Answering Questions

Whereas the Pharo language is a dynamically typed language, our participants used a variety of techniques for navigation in a codebase and for obtaining static information. This section discusses the aspects of dynamically typed language when navigating in a codebase and getting static information.

Indirect navigation. Participant P1 wished to explore the following piece of code:

```
1 box := RTBox new color: Color blue; size: 30.
2 element := box element.
3 element when: TRMouseClick do: [:event | ... ].
```

On the first line, a `RTBox` object is created. This object serves as a factory for another object called `element` in line two. Then in line three, `element` receives `when:do:` method.

He was interested in how the method `when:do:` is implemented. It is question 17 *"What does the declaration or definition of this look like?"* To answer this question, he asked for all implementations and subsequently tracked down which implementations (classes) are of interest for the case. Participant P1 expected a class containing "element" in its name. As the method `when:do:` is implemented in a superclass, he did not succeed in finding the definition of the method. Subsequently he examined the value of an `element` variable by inserting a breakpoint and running the code. The class of the value was `RTElement` — the kind of name he was expecting when searching in the list of all implementations of `when:do:` method. Thereafter he browsed the class and successively searched for its superclasses for the method implementation. P1 spent almost 2 minutes to get the correct method definition.

In the case that a method name is commonly used in a codebase, it may be inconvenient asking question 17 *"What does the declaration or definition of this look like?"* Participant P2 hesitated to query the implementation of all methods called `items`, as he expected to get an exhaustive list of methods. Moreover, he was not sure what class he should expect.

This may partially explain why participants P1, P3, and P6 commonly answered question 17 by navigating into a class definition, then into a method definition. As the list of implemented methods of the same name could be extensive when the class name was known, they could prefer to navigate to the class and then to a method implementation.

The opposite behavior was monitored with participant P4 who commonly used the ImplementorsOf tool for a quick navigation between classes and methods. As he used the ImplementorsOf tool only for navigation, he had to regularly close the unused windows.

Object creation. Participant P1 asked where instances of the `TRMouseClick` class are created. This is question 21, *"How are instances of these types created and assembled?"*. Pharo does not have a constructor (like in Java) and therefore this question does not have direct support in Pharo. Instead of question 21, participant P1 asked question 12 *"Where is this method called or type referenced?"* using UsersOf tool. The tool listed all methods, referencing the class and the participant manually checked every method to find the answer to the original question. In the case of participant P1, he got a list of 41 references, and only one method was an answer to question 21. He spent almost 2 minutes answering it.

Type identification. When the participants were interested in a variable type or a method return type, they used various strategies. Participant P3 regularly expected a type of an object based on the context he was observing, even in situations where variable names were not self-describing, *e.g.,* `l`, `lbl`. He made changes to the codebase according to his assumptions and if a test scenario worked, he went further with a particular task. In the possible case of a failure, he examined the code in more detail, e.g., variable types and method return values.

Participants P2 and P3 also made estimations on method names that a particular class should understand. In the possible case of a failure they wondered whether the behavior could be implemented elsewhere (*e.g.,* under a different method name), or was the behavior a missing feature.

4.3 Aggregated Questions

This section discusses questions which are not present in Sillito's study and are recognized in our study.

4.3.1 Aggregated Questions Related to 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 of 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's type or what is the method's return type?

In certain cases, answering question 17 *"What does the declaration or definition of this look like?"* needed to be divided into a particular sub-questions. Participant P2, first had to determine to whom the message is sent by asking question (e5) *"What is the variable's type or what is the method's return type?"* Question (e6) was typically 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. The ImplementorOf tool was used. Participant P1 expected a method definition on a particular class. Since he did not find the expected class using ImplementorsOf tool, he opened the class definition in the Browser tool asking (e1) *"Where is the method defined in the type hierarchy?"* and he searched the method in the following superclasses.

In the statically typed languages, *e.g.,* Java, question (17) is commonly answered by direct navigation from a calling method to a particular definition. In Pharo, it was necessary to perform extra steps (question e1, e6, e7) to achieve the desired information.

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 the abstract programming interface (API). Since argument name is an important API guideline in dynamically typed Pharo language, he was interested in what the argument names are in other methods: question (e2) *"What are the correct argument names of this method?"* In this case, the name was `aValueOrASymbolOrAOneArgBlock` indicating that values can be the basic ones, *e.g.,* a number or a string, a symbol (a specialized string), or a lambda function with one argument.

4.3.2 Aggregated Questions Related to Test Driven Development

In this section we discuss following questions:

- (e6) Is this entity or feature tested?
- (e7) Do the test cases pass?
- (r1) 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 generalized the question asking (12) *"Where is this method called or type referenced?"* In that particular case he found it difficult to answer the question and noted that *"this is not worth wasting time over ... writing a test should be easy"* and he wrote a new one. Later in the session, 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 (r1) *"What does the failure look like?"* are recognized by Sillito. Question (e7) could be mapped to (41) *"How can we know that this object has been created and initialized correctly?"* or (42) *"Will this completely solve the problem or provide the enhancement?"* Question (r1) 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 an exception thrown?"* Since it was difficult to identify specific questions, we used a more general form.

5. Threats to Validity

Our systematic observation of developers working with real-world codebases is performed on a relatively small number of the sessions. Given this setting, there are factors which limit the generalization of our results.

Since all the participants know one another and are concentrated in one metropolitan area — Santiago, they can share similar development techniques. To minimize the threat to validity we choose a different task for each session, and in the case of the familiar code, a different codebase.

In addition, the sessions limitation to a 40 minute time frame does not ensure that we recorded all the commonly asked questions of Pharo developers. However, our results indicate a similarity with Sillito's studies and we believe that the asked questions could be generalized to the other change tasks in Pharo. What could change is the distribution of the number of question occurrences.

The identification of specific questions based on a user behavior is non-deterministic. In some cases it was unclear what a user was looking for. The subsequent synthesis of general questions also suffer from uncertainty. We minimize the threat to validity by double checking that the sequence of identified questions make sense.

6. Summary

We replicate Sillito's work and identify similar results, comparing sessions on the unfamiliar codebase. The studies on the familiar codebase indicate deviation likely caused by different tasks and development strategies. Both missing type information and test driven development affected participant behavior and prudence on codebase understanding. Some participants have made changes based on assumptions and if a test scenario worked, they went ahead with a particular job. In the case of failure, they began to examine the code in more detail, *e.g.,* variable types and method return values.

We provide aspects and strategies on answering particular questions (Section 4.2) and a set of questions that are useful for characterizing activity related to the use of a dynamically typed language and test-driven development, questions not explicitly considered in previous research (Section 4.3).

Research question. In this study we focus on the asked questions and their occurrences, comparing them to Sillito's study. We indicate that the asked questions on the unfamiliar codebase are similar to the questions asked by Sillito. In the case of unfamiliar codebase, we observe great deviation between each session, though more work is needed.

Related work. To our knowledge there is no similar study performed on Pharo or any another dynamically typed language. Ko *et al.*, Duala-Ekoko *et al.*, and LaToza *et al.* conducted their studies on statically typed languages, *e.g.,* Java, C, C++, or C# [1, 5, 6].

Future work. In the future we expect to extend this study in three aspects: (i) to grasp a wider range of developer practices on answering questions, (ii) to analyze the difficulties the developers have when answering questions, (iii) to process data collected by the DFlow framework [7], and (iv) to provide discussion on the setting differences compared with the study by Sillito presented in Section 2.1. We could satisfy the first aspect by involving more participants. With regard to the second aspect, we will analyze existing transcription, comparing the time they spend on answering the questions and reporting the tool support.

7. Open Questions

The preliminary work presented in this paper makes a number of assumptions that we hope to verify during the workshop. Below are a number of questions we would like to raise during the PLATEAU workshop.

Making transcripts. To conduct the experiment described in this paper, we analyzed a number of programming sessions. The analysis was carried out by manually recording the screen and then playing it back to capture all the relevant parameters. Although efficient in measuring the programming activity, this is a costly analysis technique. On average, it takes us a day and a half to produce the transcript and its associated questions (general and specific) from a 40 minute-long screencast. This is hardly scaleable analyzation technique. The first question we would like to ask the audience (who has experience in this profiling programming activity) is whether there is a better approach for carrying out our analysis. *What are the procedures used to analyze programming sessions?*

Deduction of questions. Comparing it to the study by Sillito we identify more questions per session. Sillito recognizes 334 questions by analyzing 27 programming sessions. By analyzing 6 sessions, we identify 376 questions. Such a difference may be explained by the fine grain we adopted in our analysis. However, we are not sure whether this is the main reason. *What should we be careful of when defining the questions? How should we consider the differences in the results?*

Comparison of results. As we have significantly more occurrences of questions and use the percentage of the question occurrences across the four groups; see Figure 2. It allows us to compare the question distributions. But we do not know if it is the right decision. *How do we properly compare our results with previous studies? What is the recommended procedure when comparing results on exploratory studies?*

ACKNOWLEDGMENT Juraj Kubelka is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional/2013-63130188. This work has been partially funded by FONDECYT project 1120094.

References

[1] 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*, pages 266–276, June 2012. . URL `http://dx.doi.org/10.1109/ICSE.2012.6227187`.

[2] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0. URL `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

[3] N. Juristo and S. Vegas. The role of non-exact replications in software engineering experiments. *Empirical Software Engineering*, 16(3):295–324, 2011. ISSN 1382-3256. . URL `http://dx.doi.org/10.1007/s10664-010-9141-9`.

[4] S. Kleinschmager, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153–162, June 2012. . URL `http://dx.doi.org/10.1109/ICPC.2012.6240483`.

[5] A. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 344–353, May 2007. . URL `http://dx.doi.org/10.1109/ICSE.2007.45`.

[6] 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*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. . URL `http://doi.acm.org/10.1145/1806799.1806829`.

[7] R. Minelli and M. Lanza. Visualizing the workflow of developers. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4, Sept 2013. . URL `http://dx.doi.org/10.1109/VISSOFT.2013.6650531`.

[8] A. Oram and G. Wilson, editors. *Making Software*. What Really Works, and Why We Believe It. "O'Reilly Media, Inc.", Oct. 2010. ISBN 978-0-596-80832-7. URL `http://shop.oreilly.com/product/9780596808303.do`.

[9] J. Sillito, G. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on*, 34(4):434–451, July 2008. ISSN 0098-5589. . URL `http://dx.doi.org/10.1109/TSE.2008.26`.