UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PROGRAMMING AND DISCUSSING IN LIVE PROGRAMMING ENVIRONMENTS

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS, MENCIÓN COMPUTACIÓN

JURAJ KUBELKA

PROFESOR GUÍA:
ALEXANDRE BERGEL
ROMAIN ROBBES

MIEMBROS DE LA COMISIÓN:
CECILIA BASTARRICA PINEYRO
JOCELYN SIMMONDS WAGEMANN
ANDREW BEGEL

Este trabajo fue financiado por CONICYT-PCHA/Doctorado Nacional/2013-63130188, FONDECYT Regular 1120094 y 1151195.

SANTIAGO DE CHILE
2020

Resumen

La programación en vivo proporciona herramientas para consultar y modificar software en el tiempo de ejecución, sin re-compilar y reiniciar desde cero. Creemos que la programación en vivo afecta positivamente la forma que los desarrolladores trabajan y se comunican mientras cambian el software. Por lo tanto, estudiamos este fenómeno.

Reportamos 1.161 preguntas de desarrolladores formuladas durante 17 sesiones de programación en vivo. Observamos que los desarrolladores tienen preguntas similares según estudios anteriores sobre programación *no* en vivo. Sin embargo, los desarrolladores que utilizan programación en vivo observan más seguido la información en el tiempo de ejecución. Para eso los programadores exploran el software utilizando herramientas llamadas playground, inspector de objetos, y debugger. Además emplean estrategias simples para conseguir la información y optan por respuestas inmediatas después de cada cambio de software. Concluimos que las herramientas de desarrollo deberían proporcionar más opciones para acceder a la información en el tiempo de ejecución.

Luego discutimos cómo debería ser la comunicación electrónica sobre la programación en vivo. Introducimos 4 niveles de mensajes en vivo, explicando las consecuencias de los diferentes contenidos de los mensajes. El nivel 4 representa una comunicación que integra las mismas herramientas que los desarrolladores usan durante la programación en vivo.

Implementamos LightShare, una herramienta de comunicación integrada en un entorno de programación en vivo. Luego realizamos un estudio de evaluación que duró cinco semanas. Observamos que LightShare cumple su función al mejorar la fidelidad de los mensajes. Describimos el uso observado de LightShare, las limitaciones, y las sugerencias para mejorarlo, extraídas de las respuestas de los participantes.

Abstract

Live Programming environments provide tools to query and modify software at runtime, without recompiling and restarting from scratch. We believe that Live Programming positively affects how developers work and communicate during programming change tasks. We therefore study this phenomenon.

We report 1,161 developer questions asked during 17 Live Programming sessions and observe that developers have similar questions reported in prior studies about non-live programming. Nevertheless, Live Programming developers explore the runtime information more often. For this purpose, they use playground, object inspector, and debugger tools to explore the runtime information. In addition, developers make use of simple strategies to approach information and opt for having feedback as soon as possible after individual changes. We conclude that developer tools should offer more options to access runtime information.

We then discuss what electronic communication about Live Programming should look like. We introduce 4 levels of message liveness, explaining consequences of different message contents. Level 4 stands for a communication that integrates the same tools that developers use during Live Programming.

We implemented LightShare, a communication tool integrated with a Live Programming environment. We deployed LightShare and performed a five-week long evaluation study. We observe that LightShare achieves its role by improving the message fidelity. We describe tool usages, limitations, and follow-up improvements extracted from the participant responses.

Sólo se volverá clara tu visión cuando puedas mirar en tu propio corazón. Porqué quién mira hacia afuera sueña, y quién mira hacia adentro despierta.

Your vision will only become clear when you look into your own heart. Because who looks outside dreams, and who looks inside awakens.

— *Carl Gustav Jung*

Agradecimientos

Quiero agradecer al Universo por darme la oportunidad instalarme en Santiago de Chile y adquirir la experiencia de desarrollo personal y profesional que involucra optar por un grado de doctor en ciencias.

Agradecer a mis tutores, Alexandre Bergel y Romain Robbes, por estar apoyándome durante este proceso, sobre todo por confiar en mi desde el principio. También a los miembros de la comisión: Cecilia Bastarrica Pineyro, Jocelyn Simmonds Wagemann, Andrew Begel; sus comentarios fueron de mucha ayuda para mejorar mi trabajo de tesis.

Gracias a mis padres Ladislav y Katarina, a mi hermano Jaroslav y a su familia Ivanka y Tadeáš, que me apoyaron no solo durante el traslado a Chile, sino también, desde lejos, durante toda la estadía. Gracias a Olyvia Gálvez Marchant, por su apoyo incondicional durante el desarrollo de la tesis. Agradezco a Monika Fillová y a Diego Reyes Hochfarber por entregarme herramientas para el desarrollo humano que resumo en las citas de cada capítulo. Gracias a Tudor Gîrba, Andrei Chiş y Aliaksei Syrel por poder trabajar juntos, creando un nuevo y excepcional ambiente para el desarrollo de software.

Gracias a mis amigos Juan Pablo Sandoval, Milton Mamani, Pablo Estefo y Vanessa Peña, por acompañarme durante estos años. En particular, por ayudarme a instalarme en Chile y enseñarme Español. Agradezco la comprensión acogedora de Angelica Aguirre y Sandra Gaez que amorosamente y diariamente ayudan a los estudiantes. Gracias a Renato Cerro por la ayuda en el área de inglés. La lista no sería completa sin mencionar toda la gente del Departamento de Ciencias de la Computación que se encarga en diferentes puestos y deberes, manteniendo un agradable ambiente para los estudios e investigaciones.

Quiero agradecer a CONICYT, por financiar mis estudios de doctorado a través de la beca CONICYT-PCHA/Doctorado Nacional/2013-63130188, FONDECYT Regular 1120094 y 1151195. También agradecer a ESUG, Feenk y Object Profile por el apoyo brindado durante estos años.

Contents

There is only one author of this dissertation, yet plural (we did) is used instead of singular (I did) throughout the document. The effort made and presented in this dissertation is the work of at least three persons: Romain Robbes (supervisor), Alexandre Bergel (supervisor), and Juraj Kubelka (student); therefore the plural form is used throughout the document.

Chapter 1

Introduction

Nature does not hurry, yet everything is accomplished.

— Lao Tzu

In September 2018, The Stripe company, partnered with Harris Poll, surveyed more than one thousand C-level executives and more than a thousand developers and technical leaders in five countries: United States, United Kingdom, France, Germany, and Singapore [150]. They were interested in the role that developer productivity plays in a companies' success.

The report states that "*access to developers is a bigger threat to [company] success than access to capital.*" They continue that "*developers working on the right things can accelerate a company's move into new markets or product areas and help companies differentiate themselves at disproportionate rates.*" Developers are thus force-multipliers. Nevertheless, the important factor is *not* developer quantity, but how they are leveraged.

The report continues, showing that each developer spends approximately 4 out of 41 (10%) hours a week on *bad code*, equating to a loss of nearly $85 billion USD worldwide annually. In addition, the average developer spends more than 17 out of 41 (41%) hours a week maintaining code, such as debugging and refactoring. Pfleeger and Atlee estimate that developers spend about 60 percent of software maintenance on program comprehension [126].

Interaction between developers and their development tools is therefore vitally important to enable people to build efficient and effective software. Given that most software is larger than a single person can comprehend on their own, communication about the software among software developers is also of vital importance.

While many past studies looked at industrial programming languages (*e.g.,* Cobol, C, C++, C#, Java), programming environments (*e.g.,* Eclipse, Visual Studio, text editors), and communication forums (*e.g.,* Q&A Stack Overflow, mailing lists), few have focused on the industrial use of Live Programming languages along with belonging development environments and communication forums. Live Programming languages differ from other languages to the degree to which development tools are available at runtime to query and modify software programs without recompiling and restarting from scratch on every source code change. It dramatically reduces the read-compile-debug cycle time, enabling developers to be more efficient during the software produce evolution, *e.g.,* bug fixes and enhancements.

1.1. Our Proposal

We therefore believe that Live Programming is an important property that positively affects how developers work and communicate during programming change tasks. In this dissertation, we study these effects. First, we explore the kinds of questions developers ask during programming change tasks in Live Programming environments and illustrate how the questions differ from prior studies in traditional development environments. Second, we observe developer works to see how developers use Live Programming tools to answer these questions. Finally, we recognize that sharing information about Live Programming artifacts would be much easier if communication tools transmitted a much more robust interactive views of software artifacts under a discussion. We built LightShare, a software artifact communication tool that facilitates such high-fidelity sharing among Live Programming developers.

The thesis statement is:

> *We believe that Live Programming is an important property that affects how software developers work and communicate during programming change tasks, and in this dissertation, we study this phenomenon.*

1.2. Methodology Outline

In this section we briefly present the methodology we used to study the Live Programming phenomenon. The methodology is structured into three parts as an exploratory study on developer questions, an exploratory study on Live Programming usage, and a deployment study on LightShare communication tool usage. We provide methodology details separately in corresponding chapters.

Exploratory study on developer questions. Our exploratory study was conducted in June and November 2014. We studied 17 programming sessions involving 11 participants. We performed sessions on code bases to which participants were either *familiar* or *unfamiliar*. Each session took about 40 minutes and participants worked alone as it reflects their normal working habits. We transcribed all sessions and analyzed the types of questions participants asked about source code. We then analyzed complexity of those questions considering number of steps (sub-questions) they had to perform to answer them. Finally, we observed what development tools participants used during their programming sessions. This study answered the following research questions:

- ***"What developer questions do Pharo developers ask compared with the observations by Sillito* et al.*?"***
- ***"What is the complexity of answering developer questions in Pharo considering involved sub-questions?"***

- ***"How do Pharo developers use the development tools compared with the observations by Sillito*** **et al.*?"***

Exploratory study on Live Programming usage. We analyzed the same 17 programming sessions as in the previous study. We observed how participants used Live Programming tools and report episodes in which participants used their tools. We complement the documented approaches with a survey and subsequent analysis of 16 on-line programming sessions in other languages. The survey and on-line videos analysis was conducted in July and August 2017. This study answered the following research questions:

- **"Do developers use the Live Programming features of Pharo?"**
- **"How do developers use Live Programming in Pharo?"**
- **"Do developers, other than those we analyzed, behave similarly?"**

Deployment study on LightShare usage. We implemented LightShare, an electronic communication tool that improves the fidelity of shared software artifacts under a discussion. We analyzed the tool in 5-week long in-situ study that we conducted from March 12 to April 15, 2018. Prior to the study, we contacted 35 of the most active developers on a public chat service on February 23, 2018. Out of 35 developers, 22 agreed to participate in the study, and 15 of 22 of them used LightShare during the study. There were another 6 users who voluntarily used LightShare, though, we did not contact them. This study answered the following research questions:

- **"How often was LightShare used during the study?"**
- **"What information is shared using LightShare?"**
- **"What are the scenarios where LightShare was involved?"**
- **"Are developers satisfied with LightShare?"**
- **"What do they propose to improve?"**

Date	Description
June and November 2014	We conducted 17 sessions in a Live Programming environment
July and August 2017	We surveyed users of Live Programming environments and received 190 responses
July and August 2017	We analyzed other 16 programming session videos available on-line
February 23, 2018	We contacted the 35 most active users of a Live Programming environment community and asked them to participate in our in-situ study
March 12–April 15, 2018	We conducted the 5-week long in-situ study with deployed LightShare

Table 1.1: Studies timeline summary.

Table 1.1 summarizes periods when individual studies were carried out. The first two studies on developer questions and on Live Programming usage focus on developer interactions with a Live Programming environment during programming change tasks, which is the first part of the thesis statement. The third study proposes a communication tool that improves fidelity of shared information in a context of the Live Programming environment, which is the second part of the thesis statement. Detailed study setting descriptions are in corresponding chapters, including more insights about participants, their tasks, and our analysis.

1.3. Contributions

The main thesis contributions are summarized as follows and are published as shown by the references:

- *Exploratory study on developer questions.* We conducted an exploratory study on developer needs and tool support during programming change tasks while using a Live Programming development environment. We report the questions that developers ask during programming change tasks and contrast the results with a prior research in standard non-live development environment settings. We further research tool support and complexity answering the questions [101].

  We observed that our participants favored asking questions about runtime. The main reason is the fact, that they had more options to access the runtime information. To access the runtime information, our participants run examples, wrote code snippets, and retrieved objects independently of a debugger.
- *Exploratory study on Live Programming usage.* We analyzed how developers use Live Programming tools using the data of the previous study. We found that inspecting runtime application objects and executing code snippets are popular and frequently used techniques. Our participants used frequently object inspectors, debuggers, and playgrounds. We confirmed our observations in a survey and in an additional development sessions analysis [98].
- *Deployment study on LightShare usage.* We implemented LightShare, an electronic communication tool integrated in a Live Programming development environment. LightShare improves the fidelity of communicated messages by including software artifacts. We evaluated the tool in an in-situ study and reported the user satisfaction and the tool usage. In general, users were satisfied using LightShare, yet they proportioned to us challenging improvement proposals.

1.4. Scope and Limitations

This dissertation voluntarily focuses around the Pharo ecosystem, a Live Programming environment. Similarly, LightShare is also implemented in and integrated with Pharo only. Notwithstanding, Pharo is intensively used and supported by the Pharo community and industry (see Pharo Association and Consortium on `https://pharo.org`), their magnitude is smaller than the other industry standard development environments and languages.

Nevertheless, Pharo has Live Programming designed in the language and tools since its inception and its support is mature. Moreover, Live Programming is also part of the development community culture. The dissertation therefore provides valuable insights into Live Programming and how developers seek and share information while programming.

In this work we often talk about Live Programming. Since we chose to observe the Pharo language and programming environment, it would be more accurate to talk about Pharo Live Programming, or simply about Pharo. We would like to isolate Pharo and Live Programming during our analysis, being able to state which results are due to Live Programming and which results are a consequence of using Pharo in particular. On other hand, we cannot distinguish between Pharo and Live Programming concepts, because Pharo is one possible interpretation of what Live Programming means.

Similarly, prior studies report observations using particular programming languages and programming environments. Their reports are tied to the languages and tools and we can assume that developers behave similarly if they use similar languages and tools. We can expect that developers behave differently if languages and tools are different. The goal of this study is to observe whether developers behave differently when using Pharo, a programming environment that we think differs from the environments employed in prior studies.

1.5. Thesis Outline

This dissertation is structured as follows:

- Chapter 2, Live Programming, describes how the research community understands Live Programming and explores existing proposals.
- Chapter 3, Live Programming and Software Evolution, explores developer questions during a change task, while using a Live Programming environment. The question complexities and tools used to answer those questions are also reported.
- Chapter 4, The Road to Live Programming: Insights From the Practice, provides insights as to how developers use Live Programming environments (tools and features) and compares them with other industry-standard development environments and approaches.
- Chapter 5, Electronic Developer Communication, explores how developers communicate with each other while discussing software. It also introduces how the electronic communication should happen while discussing Live Programming.
- Chapter 6, Sending Software Artifacts From the IDE to the Chat with LightShare, introduces and evaluates LightShare, an electronic communication tool, integrated in a Live Programming environment.
- Chapter 7, Conclusions, summarizes the contributions of this dissertation work, discusses its limitations, and indicates our future research directions.

1.6. Other Contributions

We contributed in other ways, but due to time limitations, the other contributions are not included in the dissertation. In this section we highlight two particular works: Documenter and XDoc.

Documenter. As a part of Live Programming exploration, we also looked at how Live Programming source code can be documented. Documenter [94] is a solution that can be used to write class and method *live* comments. Such source code comments embed executable examples and executable code snippets whose results can be further explored and manipulated as Live Programming users would expect.

Documenter users can also write external documents stored in individual files. It can be particularly useful for writing tutorials, how-to documentation, and other explanatory documentation.

XDoc. The provided documentation can be then exported into a variety of formats, including HTML, Markdown, and Confluence. Those formats can be used in standardized communication services, including blogs, wiki sites, and issue trackers.

Since the exported formats are less interactive compared to Documenter, we developed XDoc [95], an *executable document* file format that holds a variety of live documents and data. XDoc files can be published and shared using standard communication media. Recipients can therefore start to read a document in a HTML format or execute the corresponding XDoc file and take advantage of the interactive documentation.

Both projects are part of a Live Programming development environment called GToolkit [28] and the dissertation author is a team member of the development team. GToolkit is an industry and research product developed by `feenk.com`. We provide more details about Documenter, XDoc, and GToolkit at the final chapter of this document.

1.7. Related Publications

In this section we provide a list of all publications and products created during the PhD work.

Main contributions. The contributions of this thesis were published as follows:

- Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and Answering Questions During a Programming Change Task in Pharo Language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU ’14, pages 1–11, New York, NY, USA, 2014
- Juraj Kubelka. Artifact Driven Communication To Improve Program Comprehension. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 457–460, 2017. Presented at Doctoral Symposium
- Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The Road to Live Programming: Insights from the Practice. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1090–1101, 2018
- Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Live Programming and Software Evolution: Questions during a Programming Change Task. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 30–41, 2019

Other contributions. Tangentially related to this thesis, the author also co-authored:

- Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Pitekün: An Experimental Visual Tool to Assist Code Navigation and Code Understanding. In *Proceedings of Jornadas Chilenas de Computación 2014*, JCC '14, pages 1–4, 2014
- Juraj Kubelka, Alexandre Bergel, Andrei Chiş, Tudor Gîrba, Stefan Reichhart, Romain Robbes, and Aliaksei Syrel. On Understanding How Developers Use the Spotter Search Tool. In *Proceedings of 3rd IEEE Working Conference on Software Visualization - New Ideas and Emerging Results*, VISSOFT-NIER'15, pages 1–5. IEEE, 2015
- Aliaksei Syrel, Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, and Stefan Reichhart. Spotter: Towards a Unified Search Interface in IDEs. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 54–55, New York, NY, USA, 2015. ACM
- Leonel Merino, Mohammad Ghafari, Oscar Nierstrasz, Alexandre Bergel, and Juraj Kubelka. Metavis: Exploring actionable visualization. In Bonita Sharif, Christopher Parnin, and Johan Fabry, editors, *2016 IEEE Working Conference on Software Visualization, VISSOFT 2016, Raleigh, NC, USA, October 3-4, 2016*, pages 151–155. IEEE Computer Society, 2016
- Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. Moldable, context-aware searching with Spotter. In *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, pages 128–144, 2016
- Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. Moldable Tools for Object-Oriented Development. In *Present and Ulterior Software Engineering*, pages 77–101. 2017
- Jhonny Wilder Cerezo Felipez, Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Building an expert recommender chatbot. In *Proceedings of the 1st International Workshop on Bots in Software Engineering, BotSE@ICSE*, pages 59–63, May 2019

Datasets. Datasets we produced and used as described in Chapters 3, 4, and 6:

- Juraj Kubelka, Romain Robbes, and Alexandre Bergel. ICSE 2018 and ICPC 2019 Research Dataset on Live Programming, March 2019. Available at: `https://doi.org/10.5281/zenodo.2591188`
- Juraj Kubelka, Romain Robbes, and Alexandre Bergel. ICSE 2018 and ICPC 2019 Session Video Recordings, February 2018. Available at `https://doi.org/10.5281/zenodo.1170460`
- Juraj Kubelka, Romain Robbes, and Alexandre Bergel. LightShare Study Surveys 2018, May 2018. Available at `https://doi.org/10.5281/zenodo.3603485`

Software. Software products that we developed during the author study work:

- Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Pitekün, an experimental programming environment for Pharo, July 2014. Available at `https://github.com/`

JurajKubelka/Pitekun

- Juraj Kubelka, Alexandre Bergel, Andrei Chiş, Tudor Gîrba, Stefan Reichhart, Romain Robbes, and Aliaksei Syrel. EventRecorder: User Interaction Collection Framework for Pharo, May 2015. Available at https://github.com/pharo-contributions/EventRecorder. The original code available at http://smalltalkhub.com, the Moose team, the GToolkit project
- Juraj Kubelka, Alexandre Bergel, and Romain Robbes. LightShare, a Discord client interagted in Pharo, April 2018. Available at https://github.com/JurajKubelka/DiscordSt
- Juraj Kubelka, Romain Robbes, and Alexandre Bergel. ICSE 2018 and ICPC 2019 Research Tools, March 2019. Available at https://doi.org/10.5281/zenodo.2591209
- Juraj Kubelka, Tudor Gîrba, Ioana Verebi, Aliaksei Syrel, Andrei Chiş, John Brant, George Ganea, Manuel Leuenberger, and Alistair Grant. Documenter: Tool For Creating And Consuming Live Documents Directly In The Development Environment, December 2019. Available at https://github.com/feenkcom/gtoolkit-documenter
- Juraj Kubelka, Tudor Gîrba, Ioana Verebi, Aliaksei Syrel, Andrei Chiş, John Brant, George Ganea, Manuel Leuenberger, and Alistair Grant. XDoc: Executable Document File Format for Live Documents, December 2019. Available at https://github.com/feenkcom/xdoc

Chapter 2

Live Programming

It's so much better to see things as they truly, actually are, not as we've made them in our minds.

— Ryan Holiday

This chapter describes our motivation for focusing on Live Programming. The reader wishing to experience Live Programming may visit the `http://livecoder.net` web site for a JavaScript example. The website provides a simple programming environment where someone can edit an application source code that runs in the background. The running application automatically changes its behavior, reflecting the edited source code.

2.1. Introduction to Live Programming

Elements of Live Programming dates back decades and are commonly understood as a programming environment that provides nearly instantaneous feedback from running programs through always accessible evaluation [123]. Alan Kay, best known for his pioneering work on object-oriented programming and windowing graphical user interface design [80], is a prominent computer scientist who focuses on Live Programming educational software [79]. Alan Kay, along with other researchers, implemented the Squeak programming language [78], an object-oriented, class-based, and reflective dialect of Smalltalk [59]. In 1996, they released Etoys [81], a child-friendly and object-oriented prototype-based programming language for use in education. More recently, Alan Kay and other researchers started the Croquet project, a 2D and 3D programming environment for collaborative work [145]. All mentioned solutions consider Live Programming as a key feature, without an intention to formalize what it means.

Tanimoto was the first one who formalized the meaning of Live Programming in terms of four levels of liveness in 1990 [153], that he subsequently extended to six levels of liveness in 2013 [154]. We explain those levels later in this section. Bret Victor challenges the meaning of Live Programming with the first talk back in 2012 [163]. He proposes that Live Programming should be like building and manipulating physical objects in the three-dimensional world we live in. It goes in hand with a fusion of the Croquet project and virtual and augmented realities. The research community needs to find out how to combine them.

Sean McDirmid [112], another proponent of Live Programming, wrote that "*the biggest challenge [...] is Live Programming's usefulness: how can its feedback significantly improve programmer performance?*" While research and industry communities believe that Live Programming *is the way to go*, there are no clear indicators that it improves (will improve) the programmer performance. The aim of this dissertation is to partially answer this question.

2.2. What is Live Programming

Degrees of liveness. Tanimoto formalizes Live Programming and classifies it in six levels, called six levels of liveness, according to the feedback that programming environments provide [154]. Each higher level extends the features of the previous one:

- *Level 1, Informative*: users must restart the whole application to obtain any kind of feedback, *e.g.,* an application that must be re-compiled and re-executed by a developer, each time users change the source code and want to see its output.
- *Level 2, Informative and significant*: developers obtain a semantic feedback on-demand, *e.g.,* by using interactive interpreters (LISP's Read-Eval-Print loops [135]).
- *Level 3, Informative, significant, and responsive*: programmers get an incremental feedback automatically after each edit operation, *e.g.,* by changing a website widget color using internet browser development tools.
- *Level 4, Informative, significant, responsive, and live*: user changes are applied to a running application without explicitly initiating the application under change, *e.g.,* by redefining a method code while an application is running (the application will execute the new code during a next method call without being restarted).
- *Level 5, Tactically predictive*: this happens, when an environment plans ahead, predicting the next programmer actions.
- *Level 6, Strategically predictive*: this happens, when an environment plans ahead and adds new behaviors that developers intend to program.

Each level comes up with a different experience and together they cover solutions such as auto-testing, Read-Eval-Print loops (REPLs), and systems where development tools and applications share the same environment [130]. Levels 5 and 6 include a degree of an artificial intelligence that induce future application changes. As this is not supported in development environments yet, we do not discuss it in detail in this dissertation.

Meaningful feedback. Hancock advocates that liveness should be meaningful [64]. Indispensable properties of the *meaningful liveness* are continuous feedback and continuous action. *Continuous feedback* ensures that users always observe a valid source of information during a program execution, *e.g.,* variable values and the program behavior changes whenever the values are changed. *Continuous action* provides the ability to iteratively alter the running application. Combining continuous feedback and action makes it easy to tune the running application values and behavior until developers achieve desired results.

As an example, Hancock compares shooting with a water hose and with a bow. The water hose provides both the continuous feedback and action. As users move the hose, they immediately see where the water stream intervenes and therefore easily hit a target. The water hose system reaches the level 4 on the Tanimoto's scale. On the other hand, the bow and arrow provides continuous feedback only: an archer shoots, the feedback is immediate (a hit or missed target). Before another shot, he or she must put a new arrow and tighten the bow again. There is no continuous action and he or she cannot easily adjust his or her previous bow strain and direction. Bow and arrow reaches the level 3 on the Tanimoto's scale.

2.3. Current Practices in Addressing Live Programming

Academia. Two research events, the LIVE programming workshop [22] and the International Conference on Live Coding [107], held their fifth and fourth editions respectively in 2019. Live Programming languages and tools were presented, most of them as experimental. Here, we provide a research focusing on Live Programming in practice.

Tanimoto presented VIVA, a visual language for image processing. Users can express an algorithm by drawing a flow diagram on a computer screen. The algorithm execution is then displayed to users by animating the diagram and image processing results. VIVA is designed on metaphors of electronic circuits and video studios to bring the level four of liveness to the image processing [153].

Burnett *et al.* described three prior implementation methods for keeping displayed data up-to-date during a program execution. Those algorithms aim to provide four levels of liveness in declarative visual programming environments without affecting the program execution performance. They analyzed how those algorithms affect the execution performance and subsequently introduced a new method that is more reliable than the others to maintain consistent response times [14].

Wilcox *et al.* were interested in whether a continuous visual feedback in direct-manipulation programming environments actually helps during programming tasks, in particular during debugging tasks. They conducted a controlled debugging experiment, contrasting live and non-live tasks and reported that participants performed twice as many changes in live tasks. On other hand, the effect in terms of time and accuracy was mixed [166].

McDirmid and Hsieh introduced a new SuperGlue language where components are constructed using object-oriented signals that represents a time-varying component states. They argue that programmers can build large interactive programs with less code compared to conventional approaches [114]. McDirmid later demonstrated with a working prototype how SuperGlue supports Live Programming (level four of liveness) [111].

Jonathan Edwards claims that the metaphor *programming is writing* is no longer helpful and proposes an alternative medium to text. The new medium in his Subtext prototype is copying of tree structures. Tree structures represent running applications and their source code in one overview, where users see both object and function definitions (source code) with their current values. If a user wants to use a `sum` function in an `employee` method called `salary`, he or she first copies the `sum` definition from one place, and pastes it to the `salary` method. Similarly, users can define recursive methods, by copying a method definition into its body. Because the current values are always visible next to the source code and updated as the source code change, Subtext supports level four of liveness [47].

McDirmid and Edwards introduced the Glitch language and programming environment which also updates running programs while a code is edited. Glitch manages application states and times, allowing replay of executions and therefore making it easier to observe how programs change [113].

DeLine *et al.* introduced Tempe, a Live Programming environment for data analysis. Tempe provides interactive and continually updating visualizations, and a query language for stored and live data. They evaluated the environment in a six month long deployment study, involving a product team at Microsoft. They observed that provided visualizations allow users to detect errors, share ideas, and monitor behavior [42].

Burckhardt *et al.* claims that Live Programming is difficult support due to the fact that imperative languages do not provide abstraction boundaries that make programming responsive and feedback comprehensible. They therefore presented a solution that separates a rendering and non-rendering aspects of application graphical interfaces. The separation allows continuous updating of application graphical representations while developers change its source code [12]. The solution was integrated in the Microsoft TouchDevelop programming environment [40].

Software programming in robotics is a complex field since the software that lives in a computer memory world blends with the physical reality in which robots moves. It is particularly difficult to write and comprehend the robotics software. Therefore, Campusano *et al.* introduced a Live Programming environment for robotics called VizRob in which users can write, execute, and debug the code [116].

Campusano *et al.* subsequently studied whether Live Programming is useful when writing and comprehending a robotic software. They measured the speed and accuracy on completing programming tasks, against a commonly used tool. They reported that there are no significant differences comparing the two tools, yet users preferred the Live Programming environment [16].

Johan Fabry, co-author of the previous study, talks about his experience conducting the user study. In his essay, he explains that while the research community needs to find a way to measure Live Programming benefits, such validation has not yet been carried out, including the study by Campusano *et al.* [16]. Fabry explains that software development in practice is complex and includes many variables that needs to be isolated during controlled experiments. The isolation unfortunately leads to rather artificial programming conditions that hinder the evaluation. The study time constraints, unrealistic code, complex external non-live libraries that had to be used in the study, and developer habits are variables that hinder Live Programming evaluations [52].

Industry. Live Programming is integrated in many frameworks and tools, particularly on the web. The Google Chrome web development tools [155] can change web pages and JavaScript code without a webpage reload. Similarly, popular JavaScript frameworks including Facebook's Reacts.js support Live Programming [156]. Reacts.js involves a declarative language, allowing update user interfaces on source code changes. Java applications can replace parts at runtime through the Java Platform Debugger Architecture [122], allowing redefine method definitions while an application is running. Apple Xcode supports interactive Swift playgrounds and customized per-object inspectors [49]. Microsoft Visual Studio 2015 allows REPLs interactive programming for C# and F# [66, 161]. Finally, Microsoft acquired CodeConnect, a startup company who developed a Live Programming extension for Visual Studio [25].

2.4. Live Programming in Pharo

For the purpose of this research, we chose the Pharo Live Programming development environment. This section describes and motivates our decision to focus on the Pharo language and tools.

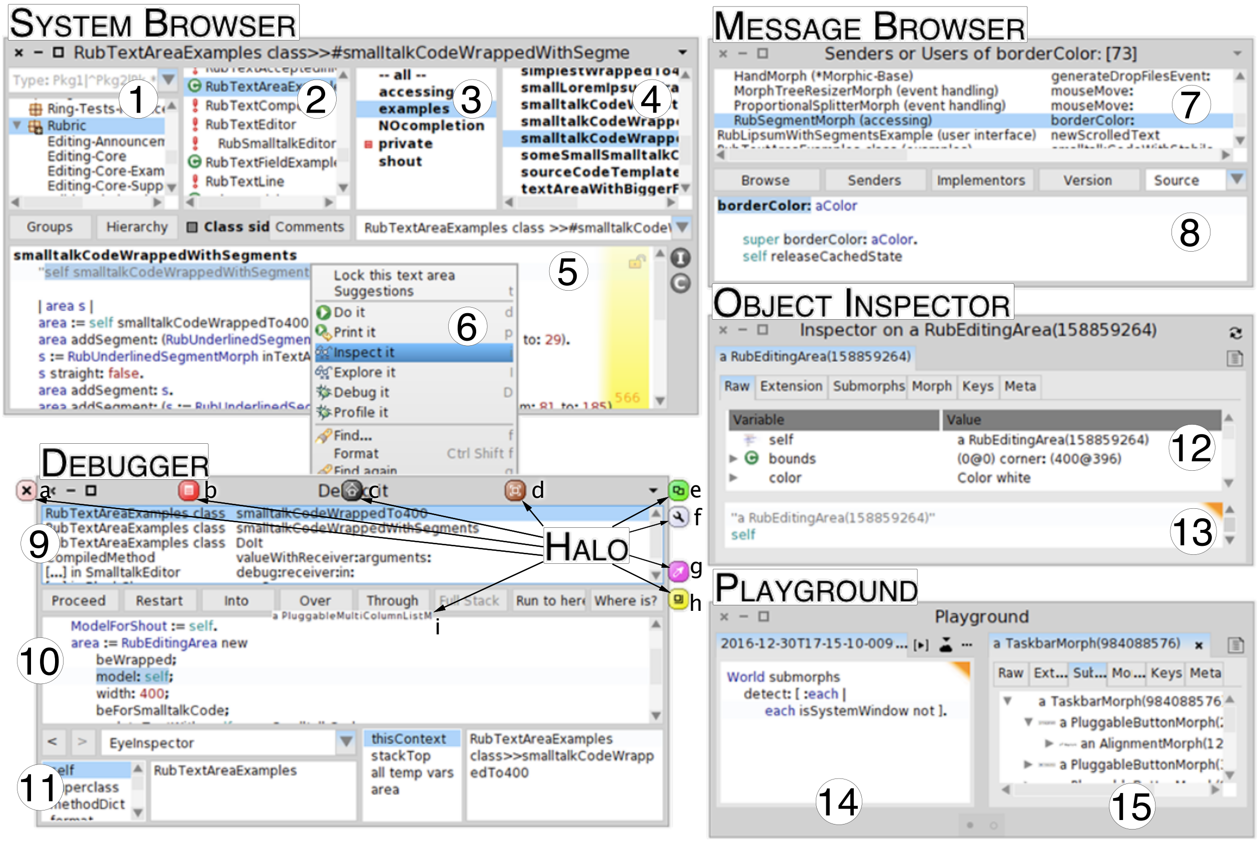
Pharo. Pharo [7] is a programming language and development environment derived from Squeak [74], a dialect of Smalltalk-80 [59]. Pharo is actively used by thousands of developers, both in academia and industry, for example: ENSTA Bretagne, JPMorgan Chase, Lam Research Corporation, Inria, feenk, CIRAD Agricultural Research For Development, Czech Technical University in Prague, to mention a few. Readers interested in who uses Pharo in academia and industry can visit the Pharo website [26] where universities, research groups, companies, and consortium industrial and academia partners are mentioned.

Pharo itself can update any part of an application at runtime. In addition, the development tools and evolved applications share the same running environment. Developers therefore operate over a running application. Developers likewise improve Pharo by altering it while it is running. Pharo thus supports level four of liveness defined by Tanimoto, which is not the case with other modern programming environments.

Pharo, as a descendant of Squeak, also features Morphic, a graphical user interface with a direct graphical widget manipulation [110]. Users can therefore select any particular application widget while the application is running, and inspect and change its internal structure and behavior. Morphic was originally implemented in Self, a prototype-based dynamic object-oriented programming language and environment [162].

We chose Pharo for this study for two main reasons. First, liveness was designed in the language and in its tools since its inception. 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 Pharo liveness features with Eclipse Java IDE [71], a traditional IDE with *very limited* liveness support (assuming a default configuration). Our aim is *not* to compare the two development environments, but rather to better explain Pharo. We choose Eclipse because it is broadly known even by occasional programmers. The remainder of this section extensively and implicitly references Figure 2.1 and Figure 2.2.



```
smalltalkCodeWrappedWithSegments
    "self smalltalkCodeWrappedWithSegment

    | area s |
    area := self smalltalkCodeWrappedTo400
    area addSegment: (RubUnderlinedSegmen
    s := RubUnderlinedSegmentMorph inTextA
    s straight: false.
    area addSegment: s.
```

HandMorph (*Morphic-Base)	generateDropFilesEvent:
MorphTreeResizerMorph (event handling)	mouseMove:
ProportionalSplitterMorph (event handling)	mouseMove:
RubSegmentMorph (accessing)	borderColor:
RubLipsumWithSegmentsExample (user interface)	newScrolledText

```
borderColor: aColor

    super borderColor: aColor.
    self releaseCachedState
```

Variable	Value
self	a RubEditingArea(158859264)
bounds	(0@0) corner: (400@396)
color	Color white

```
"a RubEditingArea(158859264)"
self
```

RubTextAreaExamples class	smalltalkCodeWrappedTo400
RubTextAreaExamples class	smalltalkCodeWrappedWithSegments
RubTextAreaExamples class	DoIt
CompiledMethod	valueWithReceiver:arguments:
[...] in SmalltalkEditor	debug:receiver:in:

```
    ModelForShout := self.
    area := RubEditingArea new
        beWrapped;
        model: self;
        width: 400;
        beForSmalltalkCode;
```

self	RubTextAreaExamples	thisContext	RubTextAreaExamples class>>smalltalkCodeWrappedTo400
superclass		stackTop	
methodDict		all temp vars	
		area	

```
World submorphs
    detect: [ :each |
        each isSystemWindow not ].
```

```
a TaskbarMorph(984088576
    a PluggableButtonMorph(
        an AlignmentMorph(12
    a PluggableButtonMorph(
```

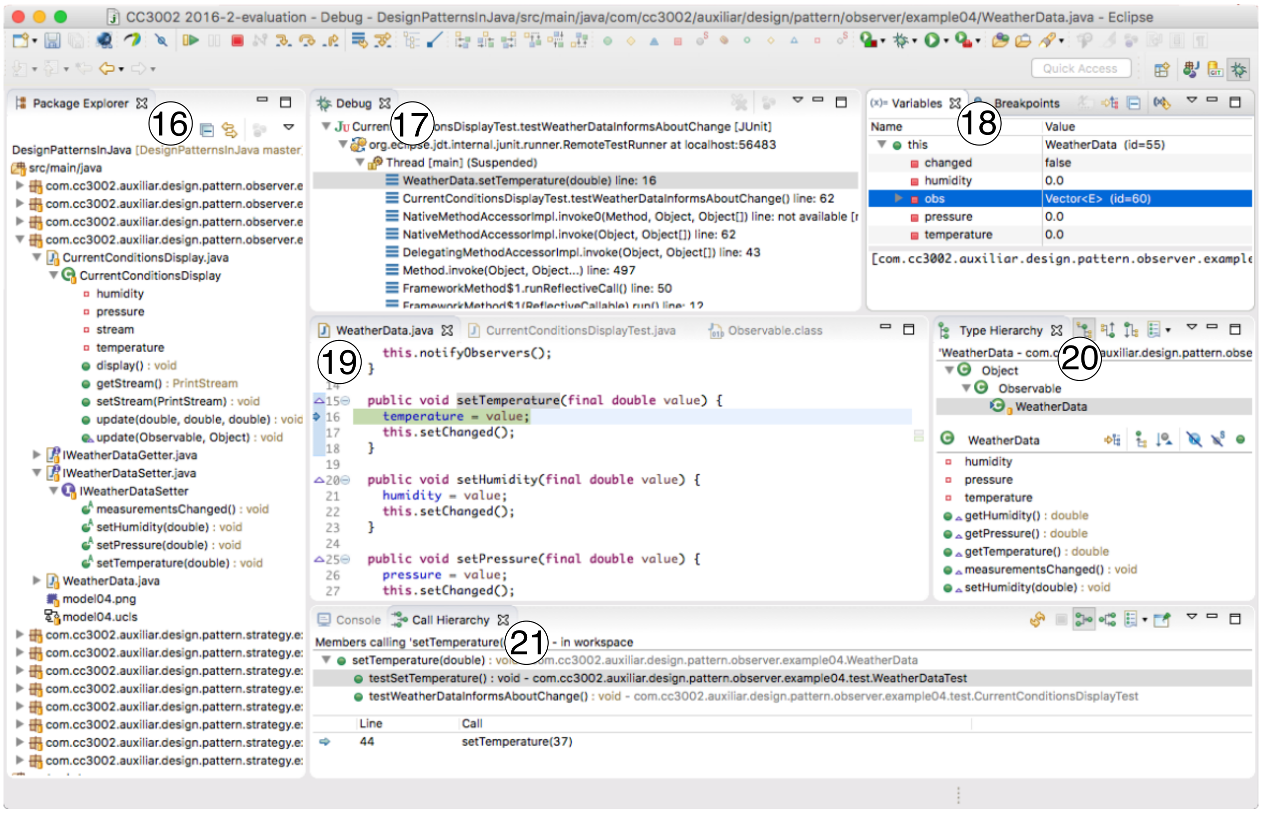
Figure 2.1: Common Pharo development tools: (1-6) System Browser, (7-8) Message Browser, (9-11) Debugger, (12-13) Object Inspector, and (14-15) Playground.

2.4.1. Source Code Navigation

Pharo offers two standard tools for static source code navigation: the System Browser (1-5), and the Message Browser (7-8). Developers can simultaneously open multiple windows for both applications.

The System Browser is the main tool for browsing, organizing and editing source code. It structures code per package (1), classes (2), method categories (3), and methods (4) on the top half of its window. It displays the source code in the bottom half of the window (5). In addition to code editing, users can perform refactoring actions, execute tests, or browse previous source code versions. The Message Browser is used for browsing lists of methods (7), be they methods referencing classes, variables, or callers or implementations of a given method. The Message Browser also includes the source code editor (8).

Eclipse features a tab-based interface (one window, many tabs), compared to window-based interface in Pharo (each tool with its own window). The source code navigation possibilities are comparable: a Package tab (16) shows the project structure and the outline of classes; a Type Hierarchy tab (20) navigates among sub- and super- classes; a Call Hierarchy tab (21) explores method execution paths; and a source code editor (19) offers a space for editing.


CC3002 2016-2-evaluation - Debug - DesignPatternsInJava/src/main/java/com/cc3002/auxiliar/design/pattern/observer/example04/WeatherData.java - Eclipse
Quick Access
Package Explorer
16
DesignPatternsInJava [DesignPatternsInJava master
src/main/java
com.cc3002.auxiliar.design.pattern.observer.e
com.cc3002.auxiliar.design.pattern.observer.e
com.cc3002.auxiliar.design.pattern.observer.e
com.cc3002.auxiliar.design.pattern.observer.e
CurrentConditionsDisplay.java
CurrentConditionsDisplay
humidity
pressure
stream
temperature
display() : void
getStream() : PrintStream
setStream(PrintStream) : void
update(double, double, double) : void
update(Observable, Object) : void
IWeatherDataGetter.java
IWeatherDataSetter.java
IWeatherDataSetter
measurementsChanged() : void
setHumidity(double) : void
setPressure(double) : void
setTemperature(double) : void
WeatherData.java
model04.png
model04.ucls
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
com.cc3002.auxiliar.design.pattern.strategy.e:
Debug
17
Variables
Breakpoints
18
WeatherData.java
CurrentConditionsDisplayTest.java
Observable.class
19
Type Hierarchy
20
'WeatherData - com.c auxiliar.design.pattern.obse
Object
Observable
WeatherData
WeatherData
humidity
pressure
temperature
getHumidity() : double
getPressure() : double
getTemperature() : double
measurementsChanged() : void
setHumidity(double) : void
Console
Call Hierarchy
21

```
CurrenonsDisplayTest.testWeatherDataInformsAboutChange [JUnit]
  org.eclipse.jdt.internal.junit.runner.RemoteTestRunner at localhost:56483
    Thread [main] (Suspended)
      WeatherData.setTemperature(double) line: 16
      CurrentConditionsDisplayTest.testWeatherDataInformsAboutChange() line: 62
      NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [r
      NativeMethodAccessorImpl.invoke(Object, Object[]) line: 62
      DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43
      Method.invoke(Object, Object...) line: 497
      FrameworkMethod$1.runReflectiveCall() line: 50
      FrameworkMethod$1(ReflectiveCallable).run() line: 12
```

Name	Value
this	WeatherData (id=55)
changed	false
humidity	0.0
obs	Vector<E> (id=60)
pressure	0.0
temperature	0.0

```
[com.cc3002.auxiliar.design.pattern.observer.example
```

```
    this.notifyObservers();
}

public void setTemperature(final double value) {
    temperature = value;
    this.setChanged();
}

public void setHumidity(final double value) {
    humidity = value;
    this.setChanged();
}

public void setPressure(final double value) {
    pressure = value;
    this.setChanged();
```

```
Members calling 'setTemperature( - in workspace
setTemperature(double) : vo om.cc3002.auxiliar.design.pattern.observer.example04.WeatherData
  testSetTemperature() : void - com.cc3002.auxiliar.design.pattern.observer.example04.test.WeatherDataTest
  testWeatherDataInformsAboutChange() : void - com.cc3002.auxiliar.design.pattern.observer.example04.test.CurrentConditionsDisplayTest
```

Line	Call
44	setTemperature(37)

Figure 2.2: The Eclipse Java development environment in a debug perspective: (1) Package Explorer, (2) Debugger execution stack, (3) Object Inspector, (4) Source Code Editor, (5) Type Hierarchy, and (6) Method Call Hierarchy.

2.4.2. Runtime Exploration

While the Pharo and Eclipse possibilities are comparable for static source code exploration, the differences emerge at runtime. Pharo offers many more options to evaluate source code snippets and access objects, as we present in the following paragraphs:

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 (6). This has many consequences in the community culture. For example, users can encounter executable examples embedded in class and method comments (the highlighted line in the 5 part), select and execute the examples (6), and observe results. In addition to the class and method example execution in the System Browser, users can write and execute code snippets the Message Browser (8), the Object Inspector (13), Debugger (10), and Playground (14). Eclipse allows a limited code snippet execution in the debug mode only (19).

Accessing objects. The object inspector tool (12, 13) displays object states in many different ways [19]. Figure 2.1 shows an inspected object (12) offering a different information presentation of the same object. The object inspector displays information about a Morphic graphical widget with the following possible views:

- The `Raw` tab displays all instance variables and their values.

- The Extensions tab exposes all widget extensions.
- The Morph tab shows the widget preview,
- The Keys tab enumerates all keyboard shortcuts that the widget handles.
- The Meta tab lists all methods and their source code that the widget understands.

Those views are configurable and differ per object type. Developers can navigate object graphs, modify instance variables, evaluate code snippets, browse the source code, and invoke object inspectors from almost anywhere. Users can open multiple object inspectors at the same time, and keep them for arbitrarily long time periods.

The Eclipse object inspector (18), displays all variable names and values associated with the current method context when Eclipse is in the debugging perspective. Developers can explore the object variables and can change values of primitive types *only*, *e.g.,* `int`. The inspector is dismissed as soon as developers leaves the debugger.

Playground. Arbitrary code snippets may be written and executed in a playground (14). Results may be explored in embedded object inspectors (15). Due to the object inspector integration, the playground usage and possibilities are equivalent to the object inspector tool described above. Users take advantage of playground when they want to execute a program (code snippet) from scratch, without any object in hand. Several playgrounds can be opened simultaneously and indefinitely. Eclipse does *not* provide a similar tool.

Debugger. The Pharo debugger (9-11) consists of a stack trace at the top (9), a source code editor in the middle (10), and an embedded object inspector at the bottom (11). 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.

When a non-existent method is invoked on an object, the debugger offers to create it, allowing the developer to implement it with live arguments and variables at his or her disposal. This process, called "programming in the debugger" can be repeated indefinitely (*e.g.,* a newly added method can in turn call a non-existing method), and is popular among test-driven development proponents.

The Eclipse debugger allows code evaluation, with *limited* possibilities to explore its results. Eclipse does *not* support multiple opened debuggers and changing source code at run-time is also limited. Eclipse users can change methods visible in the call stack. Other changes require an application restart. The Pharo debugger is invoked whenever an error occurs, while the Eclipse debugger has to be invoked manually.

Halo. Due to the Morphic [110] graphical framework integration, Pharo users can inspect and perform actions with on-screen widgets through a tool called Halo. Halo is invoked on a widget under the mouse via keyboard shortcuts. Halo buttons are displayed around the widget, activating actions such: a) deleting, b) opening a context menu with other commands, c) detaching, d) moving, e) duplicating, f) inspecting, g) changing color, h) resizing, and i) displaying its name. The name is by default denominated by the widget class name. Eclipse does *not* offer a similar functionality.

2.5. Summary

In summary, the idea of Live Programming has been present in the research and the industry communities for decades. The programming languages like Lisp and Smalltalk brought this idea to the world back in the 1980s. Two fundamental characteristics that make programming environments *live* are: instantaneous feedback from running applications; and always accessible evaluation. Tanimoto subsequently defines six levels of liveness, considering the effort and time that is necessary to obtain the feedback. The chapter concludes with the existing research in the field of Live Programming and explains why we chose Pharo for our studies, presented in this dissertation.

Chapter 3

Live Programming and Software Evolution

For there is nothing either good or bad, but thinking makes it so.

— Shakespeare

In this chapter we study 17 programming sessions and document developer questions and tool usage, answering the questions while using the Pharo Live Programming environment. We focus on developer questions during programming change tasks as we are particularly interested how developers interact with the programming environment. The study does *not* analyze other software evolution tasks, *e.g.,* software architecture, software design, and project management.

The programming sessions were conducted during June and November 2014. The preliminary results were presented at the 5th Workshop on Evaluation and Usability of Programming Languages and Tools 2014 [90]. The complete study was presented at the 27th International Conference on Program Comprehension 2019 [101] and received the ACM SIGSOFT Distinguished paper award at the conference.

3.1. 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 [143]. Further studies identified additional aspects: LaToza *et al.* investigate reachability questions [103]; Ko *et al.* unveil developer day-to-day information needs [83]; and Duala-Ekoko *et al.* exhibit questions about unfamiliar codebase [46], 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 [140] offers an extended account of the study published in IEEE Transactions on Software Engineering [143] 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 [77] of the study by Sillito *et al.* [143].

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* [154], 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 [123].

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 (for details see Chapter 2)—while the Live Programming concepts date back to the LISP [135] and Smalltalk [59] environments of the 70's.

This study offers us a critical opportunity to understand *how* liveness is used *today* in growing Live Programming environments, and offers insights to practitioners, researchers, language designers, and tool builders on how to effectively use and support Live Programming. Section 3.2 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 3.3 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 3.4 answers our research question *"**What developer questions do Pharo developers ask compared with the observations by Sillito** **et al.**?"* This question covers Chapter 4 of the analysis by Sillito *et al.* [143] and focuses on discovering the questions developers asked about their codebase. The observations are compared with the prior work by Sillito *et al.* We find that Pharo developers asked the same questions as presented by Sillito *et al.*, varying in the frequency of individual questions. In particular, questions about runtime were more frequent in our sessions. In addition, we document eight new questions that complement the existing ones by Sillito *et al.* Five out of eight questions relate to dynamically typed aspects of Pharo, three questions relate to the use of test cases.

Section 3.5 answers our research question *"**What is the complexity of answering developer questions in Pharo considering involved sub-questions?**"* We analyzed sub-questions that our participants asked to answer higher-level questions about their codebase. We find that questions about expanding focus points are simplest, while the other questions are complex, involving many sub-questions. The work by Sillito *et al.* does not cover this kind of analysis. They looked instead at how individual developer questions are supported by programming environments, which is a similar to our next research question.

Sections 3.6 and 3.7 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 section with threats to validity (Section 3.9) and summary (Section 3.10).

3.2. Background and Related Work

The aim of this section is to offer a background of the studies by Sillito *et al.* We also discuss the studies of developer questions and information needs that were made. The choice of Live Programming and Pharo is discussed in Chapter 2.

3.2.1. The Studies by Sillito *et al.*

This section describes the research by Sillito *et al.* [143] that we partially replicate. They undertook two qualitative studies, observing developers during programming change tasks.

The first study. The study published in 2005 [141] was conducted in laboratory settings with nine participants working on a code base that was new to them. Participants were graduate students in computer science. Three participants had professional development experience between two to five years. Another three participants had five or more years of the experience, and another three had no professional development experience, but did have one or more years of programming experience in the context of academic research projects. All participants had at least one year of experience using the Eclipse for Java development environment. Each participant took part in two or three programming sessions each lasting 45 minutes. They programmed in pairs. The tasks were chosen from the ArgoUML [69] issue tracking system. The project comprised roughly 60KLOC. Researchers did not expect participants to complete the task in time, but it was assumed that they would be able to make significant progress.

The researcher who was present during each session briefly interviewed participants after each session, focusing on challenges, strategy, and feelings about their progress. During each session, audio and screen recording were used, plus a log of Eclipse navigation operations.

The second study. The study published in 2006 [142] was conducted in the industrial settings with 16 participants, on code bases for which they were responsible. The participants’ professional development experience varied from four months to eight years. Sillito *et al.* monitored individual programmers, rather then pairs, because that was a normal work scenario for the participants. Some participants worked on change tasks involving C++ and Tcl programming languages and tools such as Emacs and DDD (a front end to GDB). Others worked with C# and XSLT code base using Microsoft Visual Studio. Tasks were selected by the participants who shortly explained the tasks and then spent about 30 minutes working on. Participants were asked to think aloud during the session. The size of used sowftware projects is not reported.

Participants took part in an interview after each session, focusing on the challenges and use of tools. During each session audio recording and field notes were made. The screen recording and a tool interaction log were *not* collected.

Results. Sillito *et al.* reported a list of 44 developer questions and classified them in four categories:

- The *Finding Focus Points (FFP)* category includes questions developers ask when they have little or no knowledge about source code and they need to find a source code entity (class or method) relevant to their tasks, *e.g.,* entities displaying a user dialog:
    - *"(1) Which type represents this domain concept or this UI element or action?"*
    - *"(2) Where in the code is the text in this error message or UI element?"*
    - *"(3) Where is there any code involved in the implementation of this behavior?"*
    - *"(4) Is there a precedent or exemplar for this?"*
    - *"(5) Is there an entity named something like this in that unit (project, package, or class, say)?"*
- The *Expanding Focus Points (EFP)* category covers questions programmers ask when they explore relationships among source code entities, *e.g.,* entities calling particular methods:
    - *"(6) What are the parts of this type?"*
    - *"(7) Which types is this type a part of?"*
    - *"(8) Where does this type fit in the type hierarchy?"*
    - *"(9) Does this type have any siblings in the type hierarchy?"*
    - *"(10) Where is this field declared in the type hierarchy?"*
    - *"(11) Who implements this interface or these abstract methods?"*
    - *"(12) Where is this method called or type referenced?"*
    - *"(13) When during the execution is this method called?"*
    - *"(14) Where are instances of this class created?"*
    - *"(15) Where is this variable or data structure being accessed?"*
    - *"(16) What data can we access from this object?"*
    - *"(17) What does the declaration or definition of this look like?"*
    - *"(18) What are the arguments to this function?"*
    - *"(19) What are the values of these arguments at runtime?"*
    - *"(20) What data is being modified in this code?"*
- The *Understanding a Subgraph (US)* category contains questions about understanding concepts involving several entities and relationships, *e.g.,* a behavior that particular entities provide together:
    - *"(21) How are instances of these types created and assembled?"*
    - *"(22) How are these types or objects related? (whole-part)"*
    - *"(23) How is this feature or concern (object ownership, UI control, etc.) implemented?"*
    - *"(24) What in this structure distinguishes these cases?"*

  - *“(25) What is the behavior that these types provide together and how is it distributed over the types?”*
  - *“(26) What is the ‘correct’ way to use or access this data structure?”*
  - *“(27) How does this data structure look at runtime?”*
  - *“(28) How can data be passed to (or accessed at) this point in the code?”*
  - *“(29) How is control getting (from here to) here?”*
  - *“(30) Why is control not reaching this point in the code?”*
  - *“(31) Which execution path is being taken in this case?”*
  - *“(32) Under what circumstances is this method called or exception thrown?”*
  - *“(33) What parts of this data structure are accessed in this code?”*
- The *Questions over Groups of Subgraphs (QGS)* category encompasses questions developers ask when they reason about interactions among multiple entity subgraphs, *e.g.,* how and why a program behavior changes in different scenarios:
  - *“(34) How does the system behavior vary over these types or cases?”*
  - *“(35) What are the differences between these files or types?”*
  - *“(36) What is the difference between these similar parts of the code (e.g., between sets of methods)?”*
  - *“(37) What is the mapping between these UI types and these model types?”*
  - *“(38) Where should this branch be inserted or how should this case be handled?”*
  - *“(39) Where in the UI should this functionality be added?”*
  - *“(40) To move this feature into this code, what else needs to be moved?”*
  - *“(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?”*
  - *“(44) Will this completely solve the problem or provide the enhancement?”*

Sillito *et al.* also documented number of question occurrences for all their programming sessions, anecdotes observed while answering questions, and development tool limitations. In addition, they propose several possible follow-up studies, suggesting that available tools might impact the kind and frequency of the developer questions [140]. To find out to what extent their results are valid, we partially replicate [77] the study with a different programming language and development environment: the Pharo Live Programming environment.

As the study by Sillito *et al.* published in 2008 [143] includes the two studies mentioned above [141, 142], the remainder of the thesis refers to the study from 2008, and in explicitly mentioned cases, to the PhD thesis by Sillito [140].

3.2.2. Studies on Developer Information Needs

In this section, we present other related work in less detail compared to the previous section. There are several studies that focus on developer information needs and provide a structured list of developer questions.

Kozaczynski *et al.* developed a systematic description of six question types mapped with five developer tasks that support the understanding and maintenance of complex systems [85]. Those question types are:

- What?, "What does <component> do?",
- What-If?, "What does <component> do under <condition>?",
- When?, "When does <component> do <behavior>?",
- Why?, "Why does <component> do <behavior>?",
- Where?, "Where does <system> do <behavior>?", and
- Difference?, "What are the differences between <component-old> and <component-new>?".

Ko *et al.* identify 21 questions about interaction with a codebase and co-workers [83]. 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. They determine that the most searched information included awareness about artifacts and coworkers and the most deferred search was related to the software design and behavior.

LaToza *et al.* focus on answering reachability questions and provide 12 questions rated by difficulty and frequency [103]. The following list of reachability questions is sorted by decreasing difficulty:

- What are the implications of this change? (e.g., what might break)
- How does application behavior vary in these different situations that might occur?
- Could this method call potentially be slow in some situation I need to consider?
- To move this functionality (e.g., lines of code, methods, files) to here, what else needs to be moved?
- Is this method call now redundant or unnecessary in this situation?
- Across this path of calls or set of classes, where should functionality for this case be inserted?
- When investigating some application feature or functionality, how is it implemented?
- In what situations is this method called?
- What is the correct way to use or access this data structure?
- How is control getting (from that method) to this method?
- What parts of this data structure are accessed in this code?
- How are instances of these classes or data structures created and assembled?

They identify seven developer activities (debugging, investigating, editing, reusing, reproducing, compiling, testing) 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 [104]. The list covers questions about (past or future) source code changes, source code types (classes, methods, functions, *etc.*), and type relationships. For example a debugging question is “How did this runtime state occur?”, an implementing question is “Which function or object should I pick?”, and a rational question is “Why was it done this way?” They conclude that the questions are paramount for new developing tools and programming language opportunities.

Fritz *et al.* interviewed eleven developers and present 78 developer questions with lack of tool support that involve information integration from codebases, work items, change sets, teams, comments, and Internet [56]. For the space reasons, they present 46 out of 78 questions in the paper. A complete question list is supposed to be available online; unfortunately, the provided link did not work at the time of this writing. Out of 46 questions: 7 questions are people specific (who is working on what); 20 questions are code specific (changes to codebases); 4 questions about a work item progress; 4 questions about broken builds; 2 questions about test cases; 4 questions about references on the Internet; and the last group of questions includes 5 questions that do not fit on the previous domains, *e.g.,* how a team is organized.

Duala-Ekoko and Robillard conducted a programming study on unfamiliar code and identify 20 developer questions, *e.g.,* “Is there an API type that provides a given functionality?”, “How do I get an object of type X from the typeY?” [46]. They also analyzed difficulties to answer questions and identified 5 hard-to-answer questions:

- Which keywords best describe a functionality provided by the API?
- How is the type X related to the type Y?
- Does the API provide a helper-type for manipulating objects of a given type?
- How do I create an object of a given type without a public constructor?
- How do I determine the outcome of a method call?

They observe that the major challenge to API learners is discovering relevant source code (classes) that are not accessible from the type a developer is working with. They also observe that use of Internet had no effect on the number of successfully completed tasks and time taken to complete tasks. One reason is that the participants often underestimated the time that was necessary to find an information on Internet, and adapt it into the context of a task. Consequently, they deduce implications to API design and documentation, and tool design.

De Alwis and Murphy extract 36 developer questions about source code from literature, blogs, and their own experience [39]. They classify the questions into five categories: inter-class (What calls this method?), intra-class (What methods does this type or method call?), inheritance (What interfaces does this type implement?), declarations (What class methods does this method override?), and evolution (Who has changed this element, and when?). 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. As a solution, they propose a tool that integrates information from different sources.

Roehm *et al.* conducted an observational study about software comprehension in seven companies and report on comprehension strategies [132]. For example, developers put themselves in the end user roles by inspecting user interfaces, and developers want to get their tasks done rather then comprehend software.

Treude *et al.* were concerned about the role of Question and Answer websites, such as Stack Overflow, and found that questions are centered around *how-to* questions, questions about unexpected behaviors, questions about development environments, questions about errors, conceptual questions from novices, decision help questions, and code reviews [159].

Breu *et al.* researched collaboration between developers and users and present eight question categories (missing information, clarification, triaging, *etc.*), and discuss implications for bug tracking systems [10]. They propose different user interfaces during a bug life cycle, better support for frequent questions, and a community-based bug tracking that keeps users involved.

LaToza *et al.* present developer activities and the tool support level for them [105]. Developers spend half of their time fixing bugs, 36% writing new features, and the rest of their time making code more maintainable. LaToza *et al.* demonstrate that the portions of these activities vary greatly across teams and across the lifecycle. They present developer difficulties for each activity and highlight solutions.

Smith *et al.* equipped participants with a security-oriented static analysis tool, and observed their interactions [146]. They document developer questions—including vulnerabilities, attacks, and fixes, source code exploration, and problem solving support—and implications for the design of static analysis tools. They observed that participants would have benefited from better support during program flow navigations.

Müller *et al.* investigated stakeholder information needs within a software project [118]. They present information needs for each team role and conclude that dependencies between information artifacts are important, but often not explicitly visible. As a consequence, the missing explicit information involves extra work and additional communication effort.

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.

3.3. Research Method

Participants. We had eleven male participants, including students, professors, and professional developers from distinct small local companies. The programming experience of the participants ranges 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, *e.g.,* professional and student at the same time). All but participants P2 and P5 contribute to free–software projects. Since the study by Sillito *et al.* included both students and professionals, we also included the range of experienced Pharo programmers available to us. See details in Table 3.1.

Table 3.1: Participant information.

Participant Id	Programming experience in Any language [years]	 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

As we had limited access to Pharo developers, we chose *convenience sampling* [31]. It is a type of non-probability sampling that involves participants being drawn from a population of a closely related, group of people that are easy to contact or to reach. We had two criteria of the sampling method: participants had an experience in Pharo development, and were available and willing to participate in our exploratory study.

Pilot. We conducted four pilot studies in which two thesis supervisors and the thesis author were participants. The thesis author prepared tasks for the supervisors. One supervisor prepared a task for the thesis author. All four sessions were transcribed and analyzed by the thesis author. At the beginning of the study we had only one research question "Are findings presented by Sillito *et al.* applicable to programming change tasks using the Pharo programming language?" [90]. The other research questions, presented in this and the next chapters were set out subsequently.

We decided to include all four pilot studies to the analysis for the following reasons: (i) we wanted to increase the study sample, which was difficult due to the limited access to Pharo developers; (ii) we conducted a descriptive observational study, rather than a controlled experiment for which there could be bias in favor of one treatment; (iii) the final research questions were set out after the observation sessions were conducted, which further reduced the possibility to behave unusually compared to regular programming days; and (iv) we did not observe any deviations compared to other conducted sessions.

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 Pharo software project author at that time. The pause between the two sessions was at least one day for those who participated twice.

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 they 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 [99]) and a Pharo programming environment related to a particular bug-fixing or enhancement task of Pharo and Roassal [6].

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 [108]. 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 Zenodo [99, 100, 96].

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

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 `createParent Menu: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?”*. As a result, new questions are added to one of the categories identified by Sillito *et al.*

Sub-questions identification. To identify question complexities, our session transcripts are structured to root questions (higher-level participant questions), and sub-questions that participants used to answer the root questions. Questions are therefore structured as a tree. We identified sub-questions as activities or explicitly formulated questions that were used to answer prior higher-level participant questions. For example in the transcript excerpt above, the participant P2 wanted to know how a graphical widget menu background is created. To answer this question, he first observed a particular method source code (sub-question 17), followed by another code observation (sub-question 25) for a complete understanding of the code.

Tool usage. To identify tool usage, we divide development tools into two categories: *static tools* if the main purpose is to present source code (e.g., Source Code and References Browsers); and *dynamic tools* if the main purpose is to present the state of an application (e.g., Debugger, Playground, Inspector, Test Runner, Profiler). We manually annotated each session with time slots to know how much time participants spend using those tools (considering active windows). 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.

For comparison with the study by Sillito *et al.* we also 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.

Variation. Our study is a partial replication of the study by Sillito *et al.*, described in Section 3.2.1. Here we summarize the principal differences between our study and the study by Sillito *et al.*:

- Our participants worked on tasks in the Pharo Live Programming language and environment, while the participants in the studies by Sillito *et al.* used other languages (Java, C++, C#, XSLT) and development environments (Eclipse, Emacs, and Microsoft Visual Studio).
- We defined new set of change tasks for the sessions on familiar codebases, varying in selected projects. We selected tasks on projects that are a part of Pharo, and on one external project called Roassal [6]. Sillito *et al.* tasks on unfamiliar codebase were conducted in the ArgoUML Java project.
- We recorded computer screens and audios in all sessions. Sillito *et al.* recorded screens only in one laboratory study with participants, that worked on tasks with unfamiliar codebase.

3.4. Results: Developer Questions

We collect 1,161 question occurrences from 17 programming sessions, in 13 hours of videos. On average, we report one question occurrence every 36 seconds on unfamiliar and every 47 seconds on familiar codebase. Sillito *et al.* analyzed 16.5 session hours and report one question occurrence every two minutes on unfamiliar, and every five minutes on familiar codebase.

The difference in question occurrence is given by the different methodology of recording individual sessions. In the case of sessions on familiar codebasee, Sillito *et al.* wrote questions in a paper-notebook. We recorded audios and videos for all our sessions. We also included sub-question occurrences, which was not the goal of the study by Sillito *et al.* If we contrast root-question occurrences only, we get similar numbers: 312 question occurrences in our study vs. 334 question occurrences in the study by Sillito *et al.*

Table 3.2 presents a list of all the questions, grouped by the categories identified by Sillito *et al.* The list includes new questions, not identified by Sillito *et al.*, in corresponding categories. The second column group compares tool usage 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 3.1 shows the distribution of the question occurrences among categories for unfamiliar and familiar codebases. We observe the least questions in the Finding Focus Point (FFP) category (in particular in familiar codebase sessions) and most questions in the Expanding Focus Point (EFP) category. EFP and Understanding a Subgraph (US) questions occurred about twice more in unfamiliar codebase sessions than in familiar. We detail each category in the following sections.

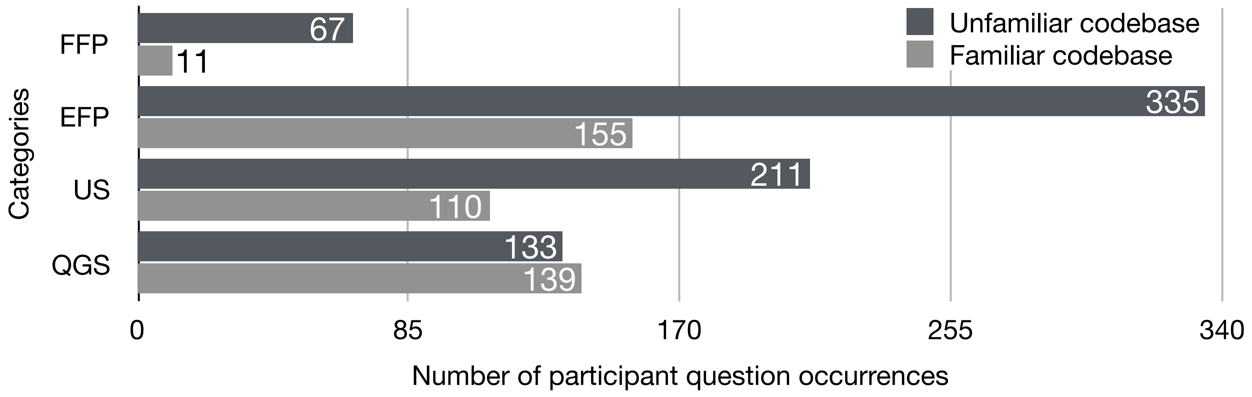


Figure 3.1: Question occurrences, including all questions mentioned in Table 3.2.

3.4.1. Finding Focus Points (FFP)

Most of our participants asked questions about finding initial points in their (familiar or unfamiliar) 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 find a class that implements it.*" (P8 in S5), and "*Does the number 46 represent the 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 3.2. 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.

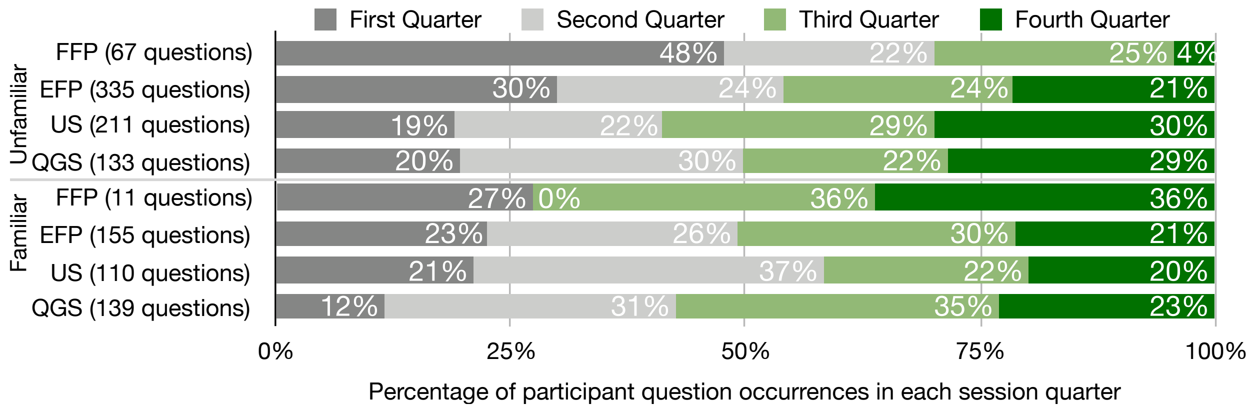


Figure 3.2: Question occurrences asked during four session quarters.

3.4.2. 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 [the] '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.

3.4.3. Understanding a Subgraph (US)

Questions in this category are about understanding of concepts that involve several entities and relationships. Participants were often interested in 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 [the] code is responsible for this behavior [box placement]?*" (P2 in S2), "*How can [text editor's content] be accessed from here [editor object]?*" (P9 in S6), 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 question 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 3.4.5. Sillito *et al.* observe question *"(23) How is this feature or concern (object ownership, UI control, etc.) implemented?"* as most frequent.

3.4.4. 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 among multiple subgraphs or the subgraph and the rest of the system. Participants asked the questions when they researched 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), and "*How is the 'position' method implemented [in two different classes]?*" (P7 in S4).

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?"* are the most prevalent questions in this category. Sillito *et al.* report questions *(42)* and *"(43) What will be the total impact of this change?"* as most frequent in this category and among all questions on familiar codebase.

3.4.5. Additional Questions

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

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 was sent by asking question *"(e5) What is the variable type or what is the method's return type at runtime?"* This 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 approximated 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 for 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 abstract method declarations. Pharo developers instead declare a "subclass responsibility" exception inside of the methods considered as abstract. If a developer forgets to override the method, the 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 implemented 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.

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.

Summary. In this section, we reported 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; and (ii) the test cases usage.

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.

3.5. Results: Developer Question Complexity

Sub-questions. Session transcripts are structured to root questions (higher-level participant questions), and sub-questions that participants used to answer the root questions. Questions are therefore structured as a tree. In the following analysis, we classify questions into four groups in order to identify question complexities: (1) root questions without sub-questions, (2) root questions with sub-questions, (3) inner-graph questions that have parent-questions and sub-questions, and (4) leaf questions that have parent questions. Figure 3.3 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.

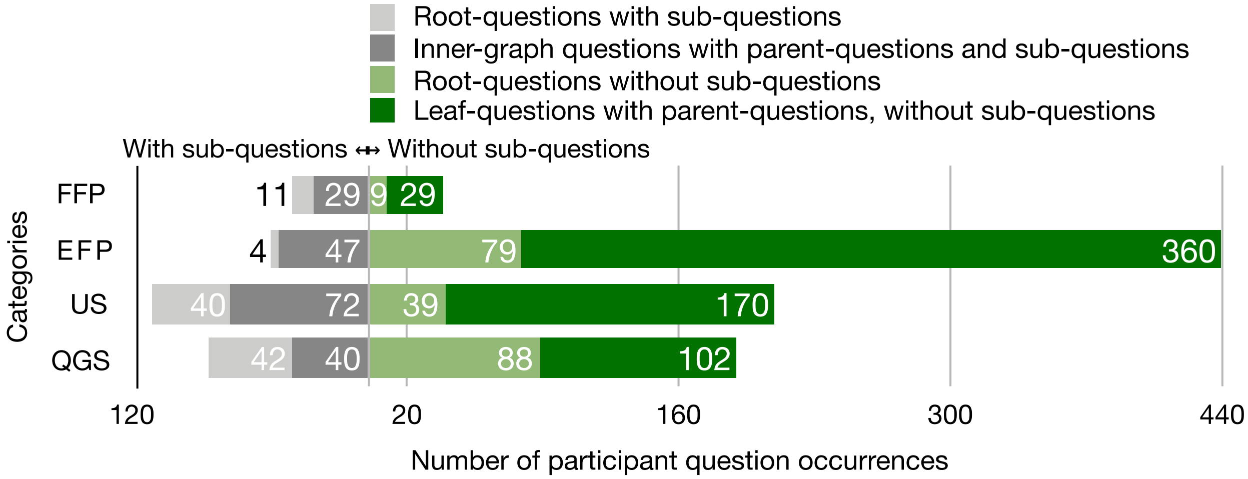


Figure 3.3: Question and sub-question occurrences classified to root questions with and without sub-questions, inner-graph questions, and leaf questions.

In the right side in Figure 3.3, we observe that 75% (876 out of 1,161) of question occurrences do not include sub-questions. Those questions are organized around categories as: 4% (38 out of 876) Finding Focus Points (FFP), 50% (439 out of 876) Expanding Focus Point (EFP), 24% (209 out of 876) Understanding a Subgraph (US), and 22% (190 out of 876) Questions over Groups of Subgraphs (QGS). Contrasting left and right sides for each question category (row) in Figure 3.3, we similarly observe that 49% (38 out of 78) FFP, 90% (439 out of 490) EFP, 65% (209 out of 321) US, and 70% (190 out of 272) QGS questions do not include sub-questions.

Participants were therefore able to answer EFP questions predominantly directly (without involving sub-questions). 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 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 3.4 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 QGS and US 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 participants picked sub-questions from all categories to answer root questions.

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

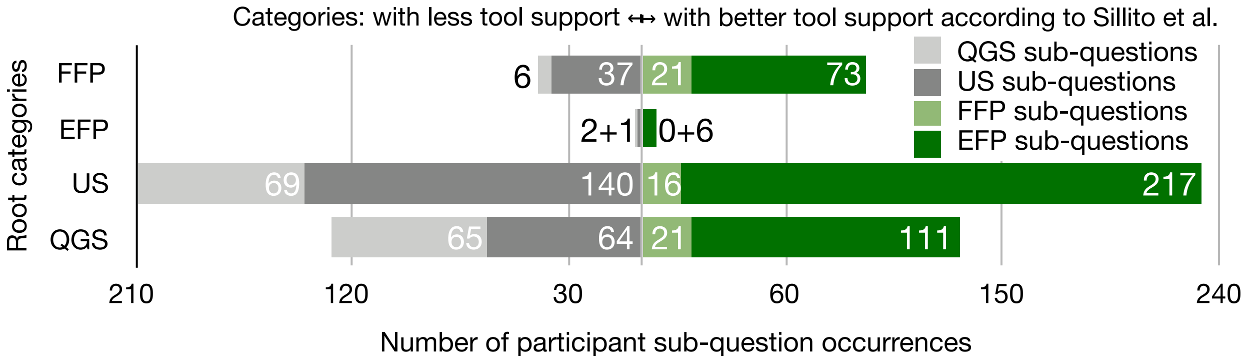


Figure 3.4: Root question occurrences classified to sub-question categories.

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 3.4.1.

Except for frequent use of EFP sub-questions, participants tend to ask 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 3.5 considers root question occurrences (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).

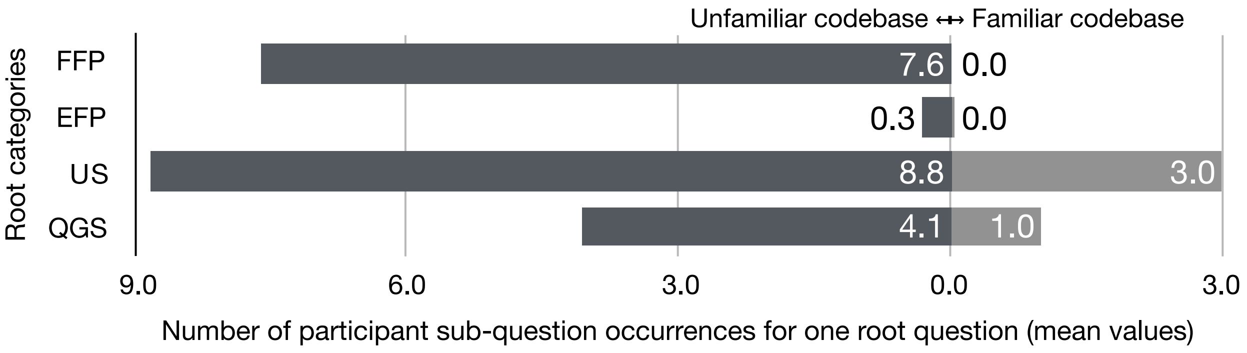


Figure 3.5: Sub-questions occurrences for one root question (mean values).

Conclusion. In this section, we present question complexity based on the number of subquestions 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 from 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.

3.6. Results: Tool Usage Answering Questions

Tool Usage. In this section, we report whether participants of the exploratory study use the Live Programming features. To this aim, we divide development tools into two categories: *static tools* if the main purpose is to present source code (*e.g.,* Source Code and References Browsers); and *dynamic tools* if the main purpose is to present the state of an application (*e.g.,* Debugger, Playground, Inspector, Test Runner, Profiler). We manually annotated each session with time slots to know how much time participants spend using those tools (considering active windows). The goal of this classification is to obtain an approximate use of Live Programming. This is only an approximation, as developers can change source code with static tools and receive immediate feedback, or use dynamic tools and observe application states without performing changes.

We match those data with time slots participants spent answering root 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 know whether participants answered questions using static (navigating source code) or dynamic tools (runtime information, *e.g.,* instance variable values). In the rest of this section, we analyze root question occurrence (312 out of 1,161).

Categories. Figure 3.6 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) root 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%) root questions.

Figure 3.7 offers a detailed look at the tool usage per unfamiliar and familiar codebase. Participants spent 27% more time using dynamic tools answering FFP root questions on unfamiliar codebase sessions. Tool usage on EFP, US, and QGS root questions was similar (varying by 6%, –8%, and 5% respectively) contrasting unfamiliar and familiar codebase sessions.

Individual Questions. Table 3.2 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

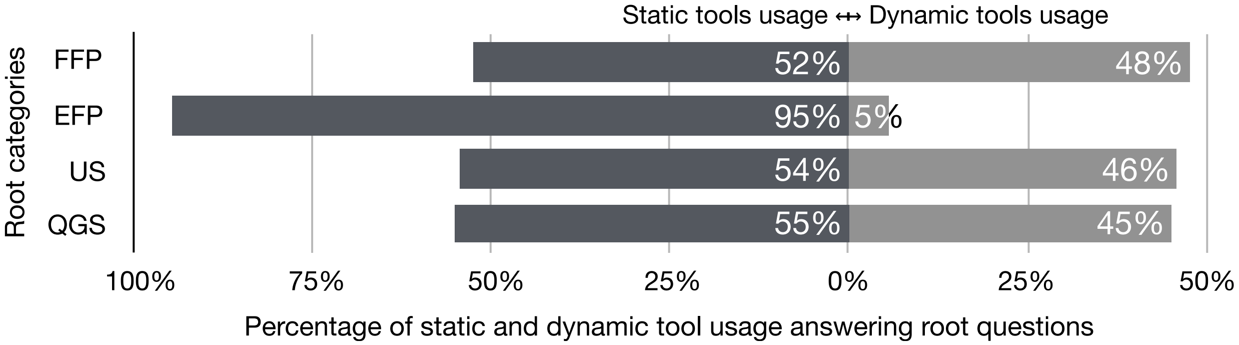


Figure 3.6: Time spent answering root questions using static and dynamic tools.

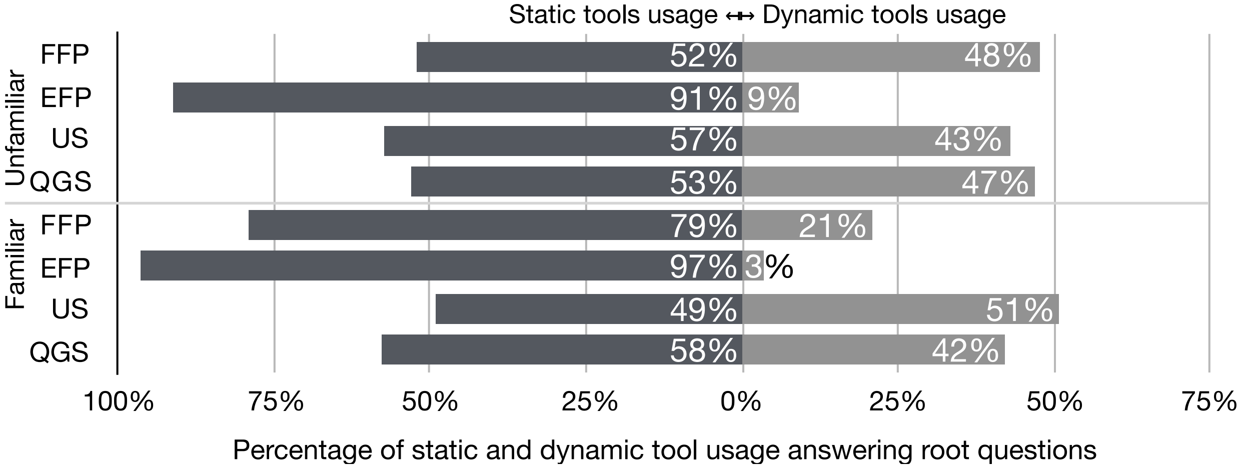


Figure 3.7: Time spent answering root questions using static and dynamic tools.

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 3.2, 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, keeping in mind, that the comparison is limited as the measurements done by us and by Sillito *et al.* are different.

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.* Our participants used often dynamic tools and we discuss particular scenarios in the following Section 3.7.

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

3.7. 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 liveness advantages and disadvantages, 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 had inspected the dialog, had copied the dialog text, and had pasted the text into the search tool he used, 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, Pharo behaves differently when an unhandled runtime exception appears: it opens a debugger with the given exception and programmers can therefore immediately explore the actual application runtime state (variables and the method-call stack), question *"(e8) What does the failure look like?"*. Moreover, they can evaluate any code snippet and edit source code inside of the debugger and if they fix the issue that raised the exception, they can accept changes and application execution can continue.

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, compared to how it was done in our study. Due to the easy access to objects, 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 to keep a proper control of application behaviors. They executed the test cases in order to check the proper application behavior and to answer question *"(e7) Do the test cases pass?"*.

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.

3.8. Discussion: Dynamic Information

Our study shows that our participants favored dynamic information. It is probably an expected observation since it is generally accepted that dynamic information is more valuable for a more accurate understanding of software. In his PhD work, Sillito mentions that some participants wrote code and used debuggers to check their assumptions.

It means that dynamic information was accessible when the study by Sillito *et al.* was conducted around 2004. Debuggers may have been a bit *rougher* around edge use cases compared to the present day, but they were absolutely available. In particular, Eclipse was released in 2001 [32]. David Flanagan in his book Java in a Nutshell from 2002 [54] describes a Java debugger. GNU Debugger was released in 1986 [33].

Sillito mentions debugger use cases several times in his dissertation. In one of his statement Sillito indicates that "*participant N2 wanted to learn about the properties of an object in the context of a particular failing case of a large loop: what I want to see is the object that is being converted to XMI [N2]. Getting a handle on this object (in the debugger, for instance) proved difficult.*" Sillito continues that "*tools used by our participants had very limited support for specifying the scope of a tool or query with the result that information displayed was noisy.*"

Unfortunately, we do not have access to more information in order to better understand why the N2 participant had difficulties obtaining the object. However, considering our observations, we think that what makes the difference is the fact that Pharo users have more options to access dynamic information compared to the study by Sillito *et al.* Pharo users can obtain objects without using debuggers. Our participants accessed the runtime information by executing examples, writing code snippets, and inspecting graphical widgets. Our participants used the runtime information to gain knowledge and confidence about source code, similarly to what is observed in the study by Sillito *et al.* In addition, our participants used runtime information even in situations where Sillito *et al.* report static analysis as sufficient.

We cannot be sure what could differ conducting our study in 2004 or conducting the study by Sillito *et al.* in 2014. Assuming that today's common programming environments make dynamic information available only through a debugger, similarly to the tools used in the study by Sillito *et al.*, we think that study differences in terms of accessing dynamic information would be similar conducting the two studies in 2004 or in 2014. On other hand, if tools used in the study by Sillito *et al.* adopt other solutions to access dynamic information, in particular Pharo's playground and inspector, we might not find differences we observed in this study. A follow-up study is necessary to confirm the hypothesis.

3.9. Threats to Validity

Empirical studies have to deal with tradeoffs between internal and external validity [139]. To deal with it properly, we decided on the following:

Internal validity. The transcript was performed by the author of the thesis. When the author was unsure of the transcription, we (the author and his supervisors) held discussions. We observed corresponding episode videos and discussed whether the episodes were correctly transcribed. During the decision-making, we compared similar episodes of our study. 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 root-questions and sub-questions based on user behaviors 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 limited choice of participants who may share similar development techniques. Our participants were all male and studies like the ones of Beckwith *et al.* [3] have shown that males are more comfortable than females with tinkering. Live Programming environments do encourage tinkering. In a follow-up study of end-user debugging [4], females were found to be less frequent, but more efficient users of tinkering, showing that the relationship is not at all obvious. As such, studies of how females use Live Programming would greatly extend this work.

Generalizing results is restrictive due to the difficulty of conducting such studies on a large scale. Transcribing one session was time-consuming and took us about two days per session. We believe that this is the reason why related research works include similar number of sessions, *e.g.,* Fritz *et al.* conducted 11 interviews [56], LaToza *et al.* run 13 sessions [104], and Sillito *et al.* carried on 27 sessions [143]. Due to the limited amount of sessions and participants, the lack of diversity in participants, tools and tasks, and the restricted session durations, it is not certain the results will generalize.

3.10. Summary and Implications

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 question occurrences in 17 programming sessions covering about 13 hours of video. The sessions were performed in the Pharo Live Programming environment. In the following paragraphs, we summarize our results about addition questions, question occurrences, question complexity and tool usage.

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. Test cases appeared in four out of seventeen sessions, and therefore, using test cases was not prevalent. Sillito *et al.* also reported that using test cases was not common in their study.

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.

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 2.4 for more details).

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. Despite 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.

In this chapter, we outlined that participants preferred dynamic information and Live Programming features. In the following chapter we will take a more detailed look at how they used the Live Programming facilities, focusing on individual tools and techniques

	Tool usage		Question occurrences in				Question occurrences in										Question occurrences in						
	By		Our sessions		Sillito ses.		Our sessions on unfamiliar code										Our sessions on familiar code						
Question Types per Category	Us	Sillito	Unf.	Fam.	Unf.	Fam.	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	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 argument 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 3.2: Tool usage and question occurrences observed in this study and the study by Sillito *et al.*

Chapter 4

The Road to Live Programming: Insights From the Practice

Peace comes from within. Do not seek it without.

— Gautama Buddha

In this chapter we study the same 17 programming sessions from the previous Chapter 3 and we document the approaches taken by developers during their work. We complement the documented approaches with a survey and subsequent analysis of 16 programming sessions in other languages, *e.g.,* JavaScript. We find that some liveness features are extensively used, and have an impact on the way developers navigate source code and objects in their work.

The programming sessions were conducted during June and November 2014. The survey and subsequent analysis of 16 programming session in additional languages were performed in July and August 2017. We presented this study at the 40th International Conference on Software Engineering 2018 [98].

4.1. Introduction

Live Programming aims to free programmers from the edit-compile-run loop of standard programming, going as far as to allow programs to be changed while they run. Recent years have shown that there is considerable interest in Live Programming, and liveness in general: the new programming language Swift from Apple features Interactive Playgrounds [49]; Microsoft recently purchased a company developing Live Programming tools [25]; Facebook's React supports instant feedback [156]; web browsers come with interactive consoles and inspectors [155]. Academically, a series of international conferences and workshops have been held on Live Programming, with LIVE 2013 being the most popular workshop at ICSE 2013 [13] (see Section 3.2 for background on Live Programming).

While some see Live Programming as "the future of programming", some of its proponents have more concrete and immediate issues. Microsoft Research's Sean McDirmid [112] wrote: “The biggest challenge instead is live programming’s usefulness: how can its feedback significantly improve programmer performance?”

While the full vision of Live Programming is yet to be implemented and made available at scale to practitioners, many elements of it are in use on a daily basis in some developer communities. In particular, many of the Live Programming concepts have been present for decades in the Smalltalk and Lisp development environments; Smalltalk and Lisp feature high degrees of *liveness*.

This offers us a critical opportunity to understand *how* liveness is used *today*, and offers insights to practitioners, researchers, language designers, and tool builders on how to effectively use and support Live Programming. In this chapter, we provide answers to three research questions on the use of Live Programming features in practice.

We provide the essential background and describe our main study subject, the Pharo development environment, in the previous chapter, Section 3.2. We therefore omit the background description here and continue with our methodology in Section 4.2. In summary, we analyzed 17 development sessions of 11 programmers described in the previous chapter, Section 3.3, totaling 13 hours of coding with familiar and unfamiliar code. This was followed by two confirmatory phases: a survey of 190 Smalltalk developers; and an analysis of 16 online videos (25 hours) of HTML/JavaScript, C#, PHP, Haskell, and C/C++ programmers.

Section 4.3 answers our first research question, **“Do developers use the Live Programming features of Pharo?”**, in the the affirmative: we find that Live Programming features and tools were used extensively by Pharo users.

Section 4.4 answers our second research question: **“How do developers use Live Programming in Pharo?”**. We describe several usages of Live Programming and contrast them against traditional programming approaches. Our overall finding is that simple approaches were favored, and their combinations can be powerful.

We then answer our third research question **“Do developers, other than those we analyzed, behave similarly?”** in two parts. First, we find that surveyed Pharo users agree with most of the observations we made (Section 4.5). Subsequently, we observe that users of other languages and tools have more limited capabilities due to the limited extent of the tool support. Despite the limitations, those users aim of getting feedback early and often (Section 4.6).

We then discuss our findings in the context of existing studies and tool support in Section 4.7. In particular, we discuss how a reduced toolset, centering on the concept of the *object inspector* supports a large range of use cases.

Finally, we close the chapter with a discussion of the threats to validity of this study in Section 4.8 and conclude with implications for tool builders, researchers, and language designers in Section 4.9.

4.2. Research Method

4.2.1. Exploratory Study

In this exploratory study we used the same data as described in Chapter 3. It includes 17 programming sessions with 11 participants. Participants were students, professors, 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.

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 Pharo software project author at that time.

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. For more details about the research method, see Section 3.3.

We printed all transcribed sessions on A0 size paper, each session on separate paper. We placed them on a wall and identified variety of approaches that participants took to make progress in their tasks. We grouped the observed approaches as follows: accessing objects, finding and modifying examples, crafting code snippets, modifying applications, unit testing, debugging, code search, and static exploration. We inferred this group based on the transcript analysis and was *not* predefined.

We used this analysis to answer our first and second research question. We complement the data with an on-line survey (with 190 responses) and an additional 16 on-line coding session videos in other languages and development tools, that we describe next. We used the on-line survey and on-line coding session videos to answer our third research question.

4.2.2. On-line Survey

Since our aim was to confirm the exploratory study findings, we used an on-line survey as it is a data collection approach that scales well [55]. The survey had three parts: demographic information, tool usage questions with multiple choices, and open-ended questions. None of the questions were mandatory. Excluding demographics questions, the survey consisted of 34 multiple-choice questions and 2 open-ended questions, and answering it took about 10 minutes. The survey is available at [99].

We first ran a pilot survey with a limited number of people to clarify our questions. We then advertised the survey through seven mailing lists related to Smalltalk communities, and Twitter. To attract participants, we included a raffle of two books.

The survey received 190 responses. Our participants were experienced (60% more than

10 years; 10% 6 to 10 years; 16% 3 to 5 years; and 14% 2 years or less), and many were from industry (44% from industry, 19% from academia, 24% both, 13% others, *e.g.,* hobby). This balances with our study participants, primarily from academia. Respondents used several Smalltalk dialects; 70% programmed daily.

4.2.3. On-line Coding Session Videos

As conducting an exploratory study is a time-consuming process even with a limited number of sessions, we complement this study with an analysis of publicly available coding session videos. We searched for programming videos on YouTube, using keywords such as "coding session", "programming session", and "live programming", and environments and frameworks that were more likely to feature a degree of liveness, such as "Javascript", "React", "C#", or "Swift". We also considered the related videos proposed by YouTube. We selected potentially relevant videos based on their title and skimming their contents. The initial list included 44 videos, totaling about 50 hours. This data is available at [99].

In the second step, we watched and categorized each video in order to determine whether it is a programming session or a pre-arranged tutorial or talk. We ended up with 16 coding session videos of about 25 hours. We then carefully watched these videos, looking for five specific events: (1) a developer observes a running application or variable values after a change; (2) a developer modifies source code or variable values of a running application; (3) a developer uses an inspector to get runtime values; (4) a developer uses a debugger to understand a problem; (5) a developer writes and runs code snippets. For each event we identified whether: (i) we observe it at least once; (ii) we observe it repeatedly after a *meaningful* source code change (programmers iteratively do small code changes and observe how it is reflected in running applications); or (iii) we do not observe it.

The video subjects use the following programming languages: 11 HTML/JavaScript (69%, 15 hours), 2 Haskell (13%, 3 hours), 1 PHP (6%, 2 hours), 1 C# (6%, 4 hours), and 1 C/C++ (6%, 14 minutes).

4.3. Results: Usage of Liveness

In order to understand how participants used Pharo tools, we first provide an overview at how often they used tools designed to display runtime information, compared to tools dedicated to work with source code. Subsequently, we provide a closer look at the individual tool usage frequency.

In the Section 3.6 we introduce two tools 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).

Figure 4.1 shows the usage of static (gray) and dynamic (black) tools over time in all the sessions. Each session label includes the following information: session ID, unfamiliar or familiar task, participant ID, and dynamic tool usage percentage. In total, our participants

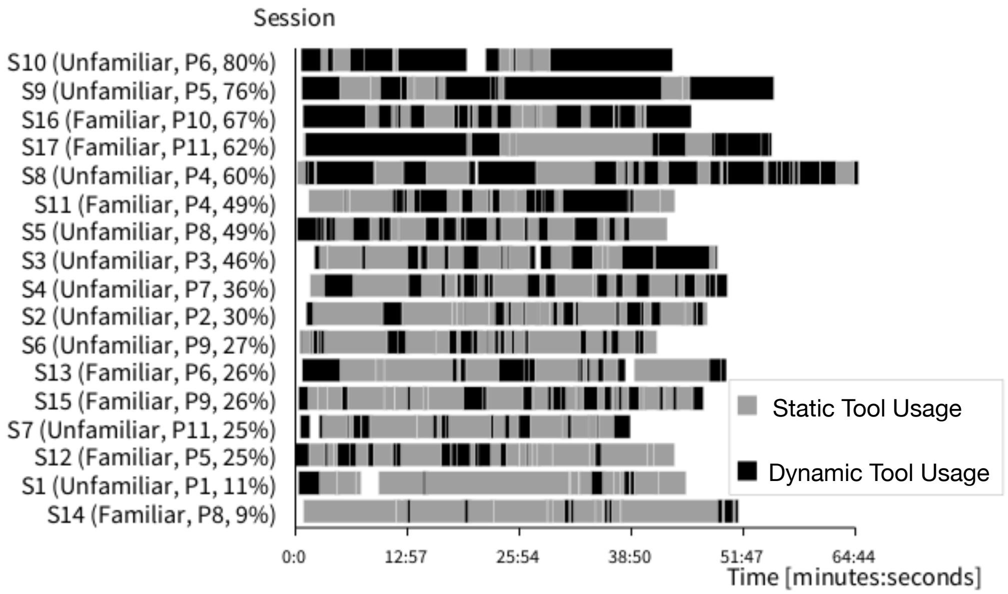


Figure 4.1: Dynamic and static tool usage per session.

spent 5.5 hours (42%) using dynamic tools and 7.5 hours (58%) using static tools, with 5 sessions having 60% or more of the time spent with dynamic tools. In addition, we can see that developers routinely switched between static and dynamic tools, even for short durations. Those numbers highlight the high overall usage of dynamic tools during the sessions, and its frequency.

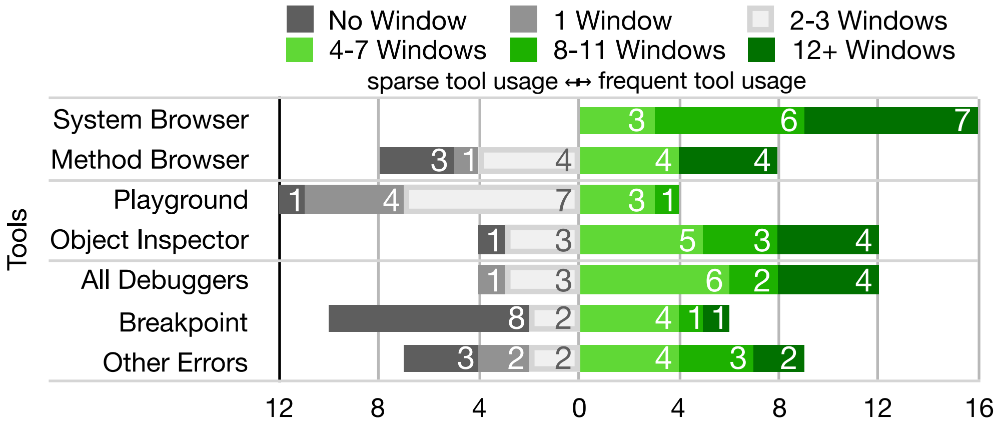


Figure 4.2: Number of sessions with sparse and frequent tool usage.

Figure 4.2 shows, for the most frequently used tools, the number of sessions with sparse and frequent tool usage. Results are aggregated for compactness: each tool is summarized by a bar divided in cells. Each cell represents the number of sessions for which a specified range of windows were opened; gray cells to the left indicate sparse or no usage of a tool, while green cells to the right indicate prolific users. We subdivide debugger usage in two categories: explicit user invocation with breakpoints, and automatic invocation by the IDE when runtime errors occur.

To better analyze the tool usage, we complement the Figure 4.2 with Figure 4.3. It shows the overall tool usage for all sessions. Each bar is divided into cells. Each cell represents one session. We separate bars into cells to highlight one particular case, that we mention in the following paragraph. When we mention number of sessions where a particular tool was used, we refer to Figure 4.2. When we present the number of opened windows, we refer to Figure 4.3.

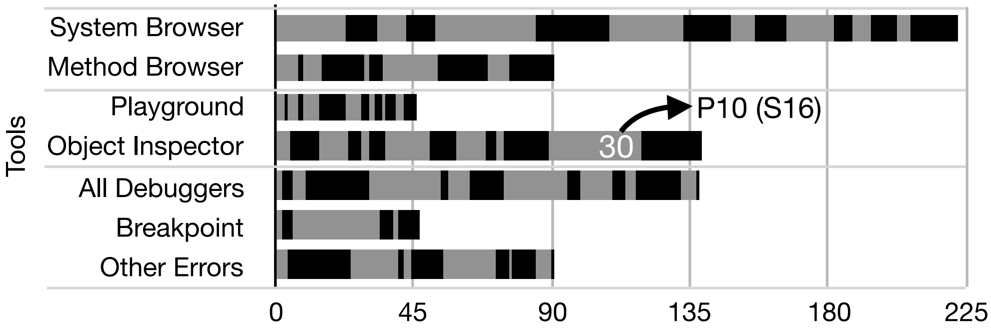


Figure 4.3: Number of tool windows opened for all sessions.

Figure 4.2 and Figure 4.3 allow us to easily rank tools by usage. The system browser (222 out of 750 development tool windows, or 30%), used to browse and edit code, unsurprisingly goes first: all subjects used it extensively. Then comes the object inspector: all but one participant used it (139 total windows, 19%), and many did very frequently. P10 (S16), our most prolific participant, opened 30 object inspector windows! Note that this is an underestimate, as this number does not include object inspectors embedded in other tools (debuggers, playgrounds). Debuggers caused by runtime errors follow (91 windows, 12%). They were present in all but 3 sessions, and appeared for the following reasons: non-existent methods (60), test case assertion failures (19), and other runtime errors (12). The method browser used for reference navigation comes next (91 windows, 12%), being extensively used by half of the participants. Next, the playground appears (46 windows, 6%), although this metric may be somewhat misleading: all but one participant used playgrounds, and these windows tend to be long-lived. Finally, debuggers invoked on breakpoints are next (47 windows, 6%), with 8 participants never using them. Other tools such as test runners (2 windows), profilers (3 windows), or version control tools (16 windows) saw little use.

Observation 5. Participants used dynamic tools frequently. Chief among them was the Object Inspector.

4.4. Results: Liveness Episodes

In this section, we describe the strategies that programmers took to make progress in their tasks. They took a variety of approaches, sometimes taking advantage of Pharo's liveness (accessing objects, finding and modifying examples, crafting code snippets, modifying applications), sometimes employing more traditional approaches (unit testing, debugging, code search, and static exploration). Each type of usage (or *episode*, reusing the terminology of Zieris and Prechelt [168]) is illustrated by one or more examples from the transcripts. Finally, we highlight how the approaches can be combined.

4.4.1. Accessing and Manipulating Objects

Accessing objects with the object inspector is the most common strategy participants took; it was present in all but one session. Object inspectors were used frequently due to Pharo's ease of executing source code anywhere, including in code browsers, playgrounds, debuggers, and inspectors themselves. Usages vary in sophistication from simple accesses (checking class names, variable values, *etc.*), to manipulation through execution of method calls and code snippets on the objects.

Basic usages. Participant P4 (S8) copy-pasted an example in a playground in order to understand a library. He had no idea what a variable was; its name "`a`" was far from useful. By adding the method call "`a inspect`" at the end of the example and executing it, P4 received an object inspector on "`a`", where he could see the object's state, including its class name.

Using multiple object inspectors. An important feature of the object inspector is that one can keep any number of inspected objects opened, for as long as necessary. We remind readers who are not familiar with Pharo, that in traditional development environments like Eclipse, developers lose access to objects when they exit a debugging session. Pharo developers contrariwise can change code and regularly execute methods in previously inspected objects to see the impact of their changes.

Participant P9 (S6) worked with the Rubric framework that creates text editors. P9 found out how the library works by opening object inspectors on objects of interest: P9 obtained an object inspector on a UI widget representing an editor view. The view had a model holding information about the contained text. Within the object inspector, P9 evaluated and asked for the view's model and opened it in another object inspector. Then P9 was sending messages to both objects, and observed how they interact and change each other.

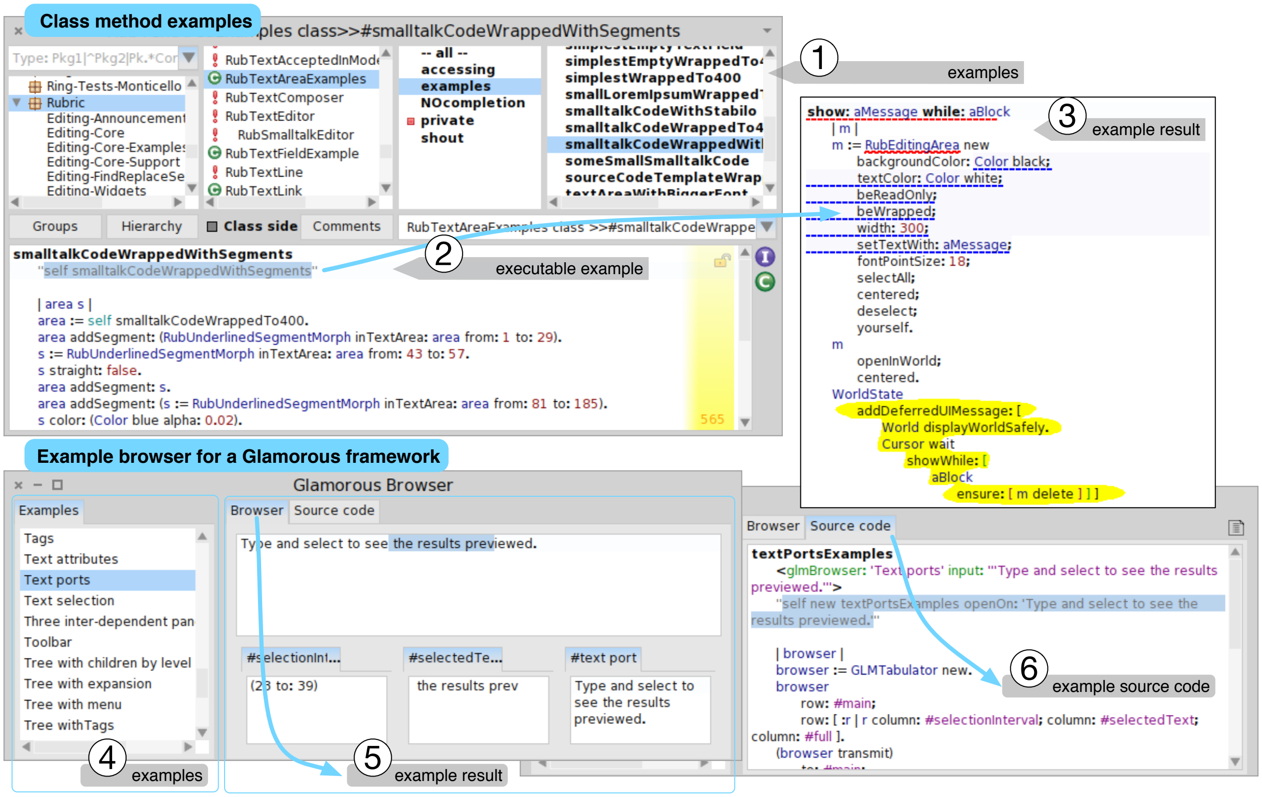
Inspecting UI objects. Participant P11 (S7) wanted to find where a mouse click that opened a UI window was handled. He inspected the UI window using a halo (see Section 2.4), and found the window's class. He then put a breakpoint in the class constructor, triggered the mouse click again, and found the source code in the debugger.

Note that being able to inspect graphical widgets does not always result in finding relevant information. Similarly to P11, Participant P5 (S9) wanted to know where a particular dialog was executed. He inspected the dialog widget, but started diving in the (complex) structure of the object, expecting to find a relevant method name (this approach is possible, but is challenging). P5 spent about 20 minutes before realizing he could do like P11 who subdivided the question in two steps: identifying the widget class, then using a breakpoint to find out where it is created.

Using multiple object inspectors with halos is possible too: Participant P6 (S10) had to fix a bug relating to an application's text editing area. Noticing that the bug is not present in another application, he used halos to find out whether the text editing areas in the two applications were similar; they were of two different UI widget classes. P6 could then focus on one class of widgets only.

4.4.2. Finding, Modifying, and Executing Examples

We define an *example* as a code snippet that can be executed and modified. Pharo developers usually pack such examples with their projects to explain how to use their code. Figure 4.4 shows two example forms present in Pharo. In the upper part, there is a class called `RubTextAreaExamples` with several example methods (1), that can be executed (2) in order to explore results (3). In the lower part, there is a dedicated example browser with a list of examples (4), with corresponding example results (5) and source code (6).



```
smalltalkCodeWrappedWithSegments
    "self smalltalkCodeWrappedWithSegments"

    | area s |
    area := self smalltalkCodeWrappedTo400.
    area addSegment: (RubUnderlinedSegmentMorph inTextArea: area from: 1 to: 29).
    s := RubUnderlinedSegmentMorph inTextArea: area from: 43 to: 57.
    s straight: false.
    area addSegment: s.
    area addSegment: (s := RubUnderlinedSegmentMorph inTextArea: area from: 81 to: 185).
    s color: (Color blue alpha: 0.02).
```

```
show: aMessage while: aBlock
    | m |
    m := RubEditingArea new
        backgroundColor: Color black;
        textColor: Color white;
        beReadOnly;
        beWrapped;
        width: 300;
        setTextWith: aMessage;
        fontPointSize: 18;
        selectAll;
        centered;
        deselect;
        yourself.
    m
        openInWorld;
        centered.
    WorldState
        addDeferredUIMessage: [
            World displayWorldSafely.
            Cursor wait
                showWhile: [
                    aBlock
                        ensure: [ m delete ] ] ]
```

Type and select to see the results previewed.

(23 to: 39)

the results prev

Type and select to see the results previewed.

```
textPortsExamples
    <glmBrowser: 'Text ports' input: '''Type and select to see the results
previewed.'''>
    "self new textPortsExamples openOn: 'Type and select to see the
results previewed.'"

    | browser |
    browser := GLMTabulator new.
    browser
        row: #main;
        row: [ :r | r column: #selectionInterval; column: #selectedText;
column: #full ].
    (browser transmit)
```

Figure 4.4: Two example forms: (i) in the upper part, examples browsed in a source code browser; (ii) in the lower part, in a dedicated example browser

Our participants found examples from a variety of sources: in documentation, in source code, in test cases, and in dedicated browsers. They were easy to execute and change, but more complex examples were more challenging to reuse. Even if seldom used, we observe that having an example browser with running (living) examples simplifies exploration and comprehension.

Basic usage. After identifying the class of the text area (see the Inspecting UI objects paragraph in the previous subsection), P6 searched for references of the text area class. Executing some of the examples he found in his search, he concluded that the problem is not in the widget itself, but likely in its configuration or usage.

Example browser. Participant P4 (S8) was tasked with enhancing the Glamorous UI framework: P4 had to show that time-consuming UI operations were in progress, avoiding the impression that the UI is frozen. P4 first searched for a code example that had such a time-consuming operation in the Glamour example browser. The example browser is a browser that only shows methods annotated as examples; the examples are automatically executed, showing the results with a domain-specific view (for Glamour, it shows the UI generated by the example). When he found a suitable example, he pasted it into a new playground to execute it. Once he was familiar with it, he made changes related to his task.

Isolation issues. Participant P9 (S6) had to replace the legacy text editor framework in the Nautilus browser with a new framework, called Rubric. He found an example of a Rubric editor. Despite having a working example, it took P9 about two and half minutes to isolate the relevant parts in order to reduce the size of the example to the minimum working solution.

4.4.3. Crafting Code Snippets

Pharo's playground is a simple but powerful combination of a code editor and an object inspector; participants used it to write small code snippets that helped them evaluate solutions and isolate bugs.

Isolating bugs. Participant P11 (S17) explored performance issues in an application. He thought that the performance issue may be caused by one of three used libraries, three application layers that were involved in the execution. To isolate the cause of the performance slowdown, he wrote a code snippet that was a minimal prototype of the application and only used the second and third layer. As the owner of the second layer library, he wrote it from memory in about two minutes. After four minutes, he had a working prototype and concluded that the third layer does not cause the performance issue.

Multiple snippets. Participant P4 (S8), tasked with enhancing Glamour's responsiveness (see the Example browser paragraph in the previous subsection), found a non-responsive example to get started. P4 constructed possible solutions, writing prototypes in different playgrounds (a total of eight during S8). Thus he thought up several solutions to the task, made changes in the library, and observed the behavior of the running prototypes. P4 said that he usually keeps examples, because "*it is an easy way to get back [to an earlier version] or share gained knowledge with others.*"

4.4.4. Modifying Running Applications

Participants frequently manipulated running applications, particularly when a feature is accessible from the GUI.

Immediate feedback. Nautilus is a source code browser in Pharo and it provides a mouse-click shortcuts to navigate method definitions and method callers quickly and directly from an observed code. P11 (S17) was asked to make the mouse-click navigation user-visible as there was no visual feedback that the feature exists. P11 worked in Nautilus itself, iteratively performed changes in the codebase, and immediately tested the change effects in the running application. This way, he verified the correctness of his progress. In this case, it was not necessary to recompile and restart Nautilus each time he made progress.

Misleading feedback. Participant P9 (S6, see the Isolation issues paragraph before the previous subsection) was also asked to change Nautilus: P9 had to replace an old text editor framework with a new one called Rubric. Upon finding relevant source code, P9 modified it to ensure that he found the right location. However, his changes were unexpectedly *not* visible in his running version of Nautilus. P9 spent about three minutes understanding why the changes were not reflected in the running application: the method P9 modified is called only once during the startup of Nautilus (this could be seen as a limitation of Pharo's liveness). P9 opened a new browser and was finally able to see the changes.

4.4.5. Traditional Approaches

Test Cases. Overall, We found few uses of test cases; they were more common on familiar codebase. Participants P4 (S11) and P8 (S14) used the test driven development (TDD) methodology: they first wrote test cases, then enhanced source code to satisfy the tests. The tests ensured their solution's accuracy. Both were working on familiar code, and knew since the start what they had to do.

P4 stated during his first session (S11) that he always uses TDD, but he did not write any test cases in session S8, on unfamiliar code. He confirmed that in this case he did not have enough knowledge to write test cases. P4 instead wrote several prototypes that reflected his ideas, and then iteratively made changes to the code that he tested on his prototypes. Similarly, participant P6 (S13) was enhancing a graphical user interface. He expressed—while repeatedly manually testing the user interface—that he usually writes test cases, but "*it is hard to write test cases for graphical user interfaces.*"

Using the debugger. Participants used a debugger at least once in each session. Only half of them used breakpoints; the others only reacted to errors. Participants did not usually change source code in the debugger. Typically, the errors had to be fixed in source code locations other than where the error occurred. P4 (S11) fixed a few test cases directly in the debugger, when it was possible to make changes locally. We did not see instances of "programming in the debugger" explained in Section 2.4 as a programming activity during which developers repeatedly add new (missing) methods, modify existing methods, and resume application execution in debuggers (without canceling the application execution).

When using breakpoints, participants were usually interested in variable values and where a method is called from. This helped them to comprehend programs (*e.g.,* P11 in S7, see Subsection 4.4.1, paragraph Inspecting UI objects). P8 (S5) used the debugger to find where and how a particular application part is initialized. Later, he regularly changed the code, executed it, and watched if the implemented parts worked and what was necessary to fix next. He rewrote his code several times. One reason was that the objects he needed were available at different moments of the execution and he therefore gradually refined his overall knowledge as he proceeded in his task.

P6 (S10) spent nearly the entire session in a single debugger, exploring in detail a long method with several conditionals. He carefully read each statement, evaluated expressions to check their return values, and observed all variable values. Finally, he identified where to make the change, and changed it (in the debugger itself).

Text Search. Along with reading source code statically, search is a common tool in traditional IDEs. On the contrary, we observed few text searches. A possible reason is that Pharo has fragmented search tools, that increase the cost of searching: one tool searches in source code text, while others do semantic searches *e.g.,* class and method names.

Participant P5 (S12) looked for a class implementing a tree graph, searching for classes including the word *tree*. He was not successful, but was convinced that "*something as fundamental as a tree object structure has to exist*". He thus searched on the Internet and in an online book. He did not find anything and solved his task differently.

P6 (S13) wanted to write and execute an one-line code snippet. He wrote a class name and subsequently wanted to write a method name whose exact name he did not remember. He was typing different possible names while observing what auto-completion suggests, and stating "*there should be a method named something like open.*" After several tries he found the method `openWithSpec`.

Source code exploration. When all else failed, participants resorted to static source code exploration. Deciding when to stop reading code and start changing it was not a trivial task.

When stopping too early, one risks missing important information and making incorrect decisions. P2 (S2) was on the right track at the beginning of his session. Before any changes, he first statically explored code in order to confirm his thoughts. He focused on the right code, but dismissed details because he thought it was "*too low-level code.*" He could not confirm his assumptions, missed important code for his task, and ended up with a complex solution.

Exploring more source code than necessary neither lead to good decisions. P1 (S1) spent his session mostly browsing code. He soon ended up in a framework that was not important to understand; he spent almost half his time exploring it. While he gained a comprehensive knowledge about it, he did not find a starting point for the task.

4.4.6. Combining Approaches

These episodes are not observed in isolation. Participants switched techniques during the course of a session as necessary, and also combined techniques, thanks to Pharo's ability to open unlimited instances of each tool. For instance, P11 combined halos, object inspectors and debugging in S7 (see Subsection 4.4.1, paragraph Inspecting UI objects). This is best exemplified by the participant P7, while seeking to isolate a bug in a graphical library:

P7 (S4) wrote a code snippet in a playground. The code represented the minimal steps to reproduce the bug in the application. In addition to running the application, P7 was concerned about a particular object it used. He thus added the `inspect` message to his snippet and executed it, obtaining the application's graphical window and an object inspector with the object of interest. He then crafted a similar code snippet that involved other UI widgets, and behaved correctly. He executed this second snippet, and opened a second object inspector. P7 regularly modified and executed the code snippets on both cases and observed the differences in behavior. By observing both running examples and the corresponding source code, he was able to reveal the problem and fix it.

Observation 6. Participants employed a variety of approaches, but preferred simpler ones, e.g., *object inspectors and debuggers. Combining approaches,* e.g., *writing code snippets and inspecting objects, had powerful results.*

4.5. Results: On-line Survey

To confirm our exploratory study observations described on the previous pages, we created a questionnaire that we distributed in Pharo and other Smalltalk communities. We received 190 responses that we present in this section. The survey is available at [99].

Frequency answers. Figure 4.5 displays an overview of frequency answers. We asked for specific tool usage and practices. In particular, we asked what activities they perform while having object inspectors, debuggers, playgrounds, and running applications. Additionally, we asked where they look for examples. For easy reading, we simplified the wording of the individual claims. For easier comparison of the claims, the responses “Rarely”, “Occasionally/ Sometimes”, “Regularly” are aligned on the left, whereas “Often” and “Very often” on the right.

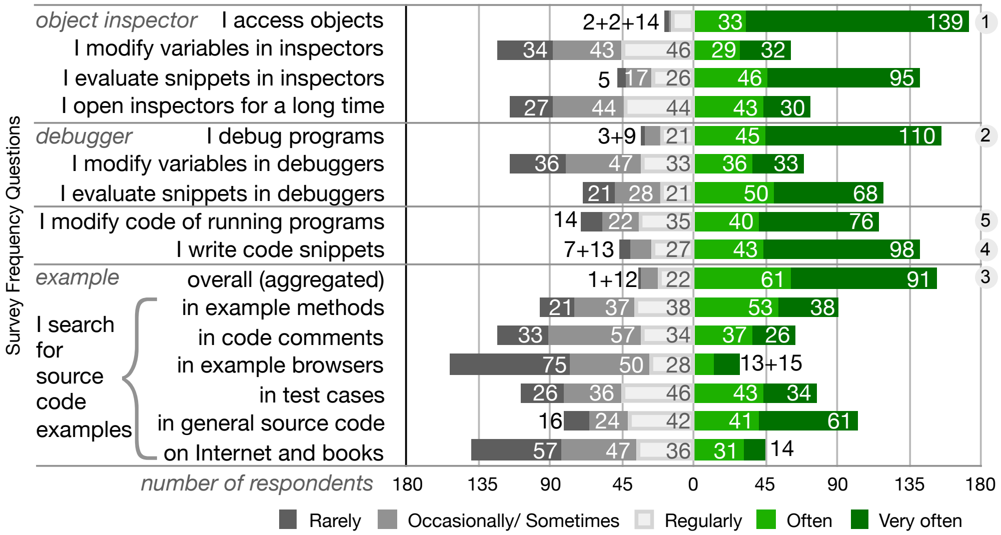


Figure 4.5: Survey frequency questions and answers.

Tool usage. We find that our respondents indicate that they use the liveness in Smalltalk IDEs overwhelmingly and often. All but 3 of the respondents (98%) indicated that they used at least one of the tools often. The object inspector once again took the lead (172 respondents used it at least often, 90%), followed by the debugger (all debugger usages, 155 respondents, 82%), then examples (aggregated, 152 respondents, 80%), code snippets writing tools (*e.g.,* Pharo’s playground, 141 respondents, 74%). Modifying running applications was somewhat less popular (116 respondents, 61%).

Specific practices. We note that keeping object inspectors open for long periods of time (more than one hour) is practiced often by a respectable amount of our respondents (73 respondents, 38%, Figure 4.5), and regularly by an additional 44 respondents (23%). Participants also tended to evaluate code snippets more than directly changing data structures, be it in the object inspector or the debugger (74 vs 32%, and 62% vs 36%). This also indicates that the playground is not the only place where code snippets are crafted. We note that examples were popular overall, but that the sources are varied. Dedicated example browsing tools, still relatively new, were not used extensively.

Agreement answers. Figure 4.6 reveals agreement responses. The survey includes statements reflecting our exploratory study observations. For easy reading, we simplified the wording of the individual statements in the figure. For easier comparison of the claims, the responses "Disagree", "Somewhat disagree", and "Neutral" are aligned on the left, whereas "Somewhat agree", and "Agree" on the right.

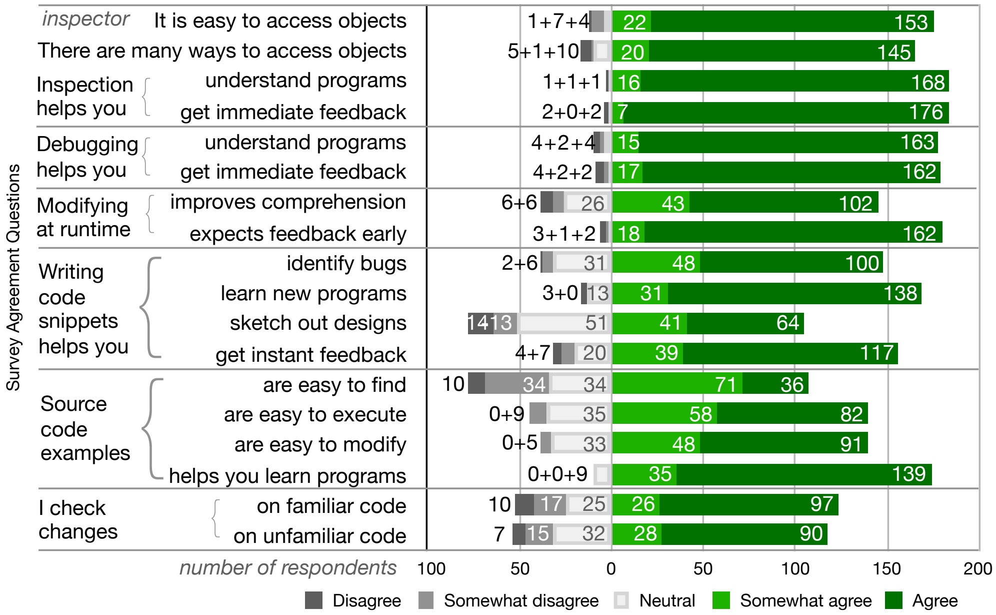


Figure 4.6: Agreement answers.

We find that our respondents generally agreed with the statements we proposed. There is a particularly broad agreement with our assertions on the object inspector, and the debugger. Agreement is more moderate in the case of examples (finding relevant source code examples is not always easy), and code snippets (code snippets are less used to sketch out program structures). Respondents found that fast feedback was somewhat easier to achieve in object inspectors and debuggers than with code snippets (presumably since many queries in the object inspector or debugger do not involve writing code). They finally indicated a strong preference to check their changes as often as possible, and not only after finishing a task. We restate that all of the statements proposed were generally met with agreement.

Comparisons. We also asked our participants to compare reading code with other approaches, on both familiar and unfamiliar code. Participants indicated that they found it easier to read code than to write either tests or code snippets. On the other hand, they indicated that they found it easier to understand programs by running them (or running examples), rather than reading code. In all cases, reading code was slightly easier when said code was familiar.

> *Observation 7. Survey participants confirmed that they used liveness extensively; the object inspector was again the most frequently used.*

4.6. Results: On-line Coding Sessions

To confirm that developers in other languages and development environments also seek for the liveness features, we analyzed on-line coding session videos. This section presents results of 16 coding sessions of about 25 hours. The video subjects use the following programming languages: 11 HTML/JavaScript (69%, 15 hours), 2 Haskell (13%, 3 hours), 1 PHP (6%, 2 hours), 1 C# (6%, 4 hours), and 1 C/C++ (6%, 14 minutes).

Figure 4.7 displays our observations for each programming approach. For easier comparison, the conclusions "Not Observed" are aligned on the left, whereas "Appeared" and "Iterative" on the right.

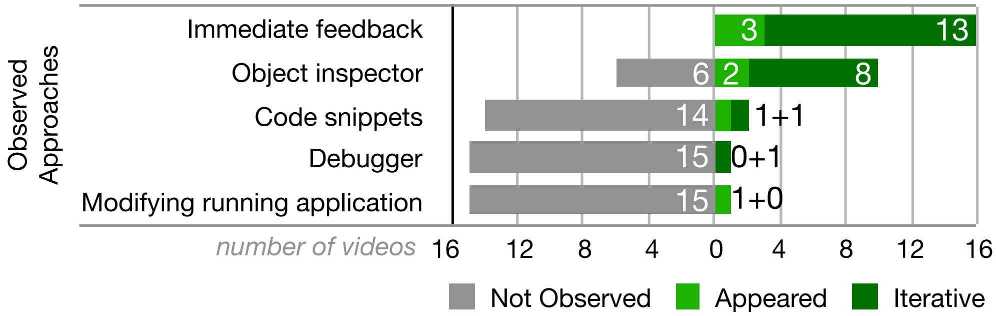


Figure 4.7: On-line coding session video observations.

Immediate feedback. This was the most used approach. Participants usually tried to write the minimum amount of source code that they were able to run and test; sometimes an incorrect behavior was expected. In some cases, the application restart was performed automatically, others manually. Two programmers used test cases that were re-executed automatically in a text console.

Object inspector usage. An object inspector, in particular HTML/DOM inspector and JavaScript object inspector, was the second most frequent tool. Developers predominantly inspected HTML pages and explored HTML DOM and CSS rules. JavaScript's object inspector was frequently used in a browser console in order to check a variable value, less often to observe an object's internal structure. Two programmers expressed confusion when they explored the internal structure of JavaScript objects: the JavaScript inspector exposes many internal object attributes (*e.g.,* __proto__ variables) that are usually not relevant to developers.

Code snippet usage. We observe two sessions where programmers wrote and executed code in a JavaScript Internet browser console. In the first case, the developer was testing return values while sending methods to an object. In the second case, the developer wondered if sending a particular method to an object was properly handled. To obtain the object, he logged the variable to the console in his source code and refreshed the application. When the object appeared in the console, he used the "Store as Global Variable" feature to make the object accessible across runs. He then resumed his work, sending the method to the object several times.

Debugger usage. The debugger was used in only one session, where a participant put breakpoints in a web browser and observed variable values of his recently written program.

Modifying running application. One programmer modified the runtime state of his JavaScript program. He evaluated some code in a browser console and thus tested its correct behavior.

Discussion. We conclude that developers use feedback in their runtime environment; this includes executing applications and occasionally runtime state exploration. We think tool support has an important impact. We believe that participants used the HTML/DOM inspector iteratively because it is a mature solution: they can easily jump from HTML to source code, manipulate it and thus understand how it works. Similarly, they frequently used the inspector integrated in web browser consoles. But developers did not explore internals of JavaScript objects: understanding it is complex and tool support is insufficient. Further analysis is needed to confirm these implications and is out of scope of this thesis.

Observation 8. Developers in other languages use feedback when they can have it; more limited tool support restricts their options.

4.7. Findings and Discussion

We discuss our findings taking as our main point of comparison the study of Sillito *et al.* [143] and in particular the extended account found in Sillito's thesis [140]. This is the most extensive and in-depth account of a programming observation study that we know of. It also has the richest source of implications to which we can contrast findings. We also explicitly refer to other work as necessary.

4.7.1. On Liveness

No silver bullet. We agree with Tanimoto [154] that "*Liveness by itself is not a panacea for programming environment. Neither it is necessary for all programming tasks*". We find some instances where subjects were unable or unwilling to use liveness; they were subject to the same static exploration issues observed in traditional IDEs. Examples include P1 spending too much time in an overly exhaustive static exploration, and P2 missing valuable information from a too shallow one (similar to the "Inattention Blindness" phenomenon described by Robillard [131]). Missing knowledge, as observed by Murphy-Hill *et al.* [120] was a factor too, as we observed several instances where the halo or the debugger could have been used, but they were not.

More generally, exploring unfamiliar code also imposed restrictions on how participants could obtain feedback, particularly with test cases (P4, P6), echoing observations by Sillito *et al.* who states that writing test cases was the exception and in some cases challenging. In a study by LaToza *et al.*, they who found that tests were not extensive enough and that developers could not rely on them [103].

In other cases, participants employed a variety of liveness techniques, but were not successful. P9 expected feedback, and lost time when he did not receive it. P4 kept several prototypes of solutions to his task as code snippets, experimenting with them iteratively, but did not succeed in his task as he "*did not want to play with threads, because it is too complex*", despite the necessity of threads.

Liveness eases checking assumptions. Checking assumptions is important to ensure task progress, as Sillito observed: "*Programmers' false assumptions about the system were at times left unchecked and made progress on the task difficult.*" This was more likely when questions could not be answered directly by tools. In particular, Sillito found that setting breakpoints in the debugger was often used to assess if code was relevant for the execution of a feature. LaToza *et al.* [102] observed that participants "gambled" when they did not have sufficient information, and noticed that the "false facts" they inferred often led to further chains of "false facts".

Our participants were able to check their assumptions in a variety of additional ways, such as P11 quickly discarding an application layer from consideration thanks to a code snippet; or P6 finding out that two widgets were from different classes by inspecting them, and then finding functioning examples of the widget to infer that the error came from its configuration, rather than the widget itself.

The participants were often able to quickly check their assumptions, ranging from the simplest (what is the class of this object?) to the more refined, thus spending less time on false tracks. When it was possible to progressively perform source code changes and test the behavior on running applications, participants did take full advantage of it. For example, participant P11 tested his work after *every small change.* Sometimes he expected a broken behavior, sometimes a positive progress.

Liveness eases refining context. Context is a major hurdle that tools do not handle well, according to Sillito: "*A programmer's questions often have an explicit or implicit context or scope. [...] Tools generally provide little or no support for a programmer to specify such context [...], and so programmers ask questions more globally than they intend.*" The broadened scope makes information more likely to be missed and makes it even harder to check assumptions.

Pharo's static exploration tools have the same scoping issues as traditional tools. In fact, due to the lack of type annotations, scoping issues are worse than in statically typed languages [65, 51, 125]. However, we have seen the use of liveness features to reduce the scope of investigations, such as P11 discarding an application layer thanks to a code snippet. A key point is that Pharo makes it easier to scope runtime questions as it is often *not necessary to run the entire application.* A developer can focus on an application's objects of interest (e.g., P9 inspecting a model and its view) and explicitly interact with those, for as long as necessary. Similarly, code snippets can call *parts* of existing code, and examples focus on a subset of an API.

Some "hard questions" are easier to answer using liveness. Sillito describes 44 questions developers asked during his sessions. Questions in categories 1 (finding focus points) and 2 (expanding focus points) all had full or partial tool support. Questions in categories 3 (understanding a subgraph) and 4 (questions over groups of subgraphs) had mostly minimal, or at best partial support. These questions in categories 3 and 4 were higher level and the hardest to answer.

While liveness can not help for all questions, in some instances it definitely can. Question 27 (how does this data structure look at runtime?) is directly supported by the object inspector. Sillito found that 15% of sessions included this question. All but one of our sessions (94%) used the object inspector. Similarly, Question 37 (What is the mapping between these UI types and these model types?) can be easily answered by inspecting UI elements, thanks to halos.

Several questions in the fourth category concern comparisons between types, methods, or runtime behavior. Sillito states that "*Generally making comparisons is difficult*" and "*In some cases, even harder is a comparison between a system's behavior in two different cases as in question 34 (How does the system behavior vary over these types or cases?)*". One of Sillito's subjects wanted to "*figure out why one works and one does not*", and did two series of static reference navigations, comparing the differences. The participant obtained only a partial answer and missed the most important difference.

We found these kinds of comparisons in 12 out of 17 sessions, often thanks to multiple object inspectors or snippets. Examples include P4 experimenting with multiple solutions in parallel, P6 inspecting two similar widgets for differences, P7 contrasting the behavior of two code snippets (one correct, one incorrect), or P11 contrasting an application with a snippet. While it is possible to open multiple debuggers at the same time, we did not observe this.

4.7.2. On Tool Support

Tool support drives Live Programming. Some major differences between this study and Sillito's stem from tool support. Sillito states "*[...] the questions asked and the process of answering those are influenced by the tools available [...]. Given a completely different set of tools or participants our data could be quite different.*"

We found evidence of this, as behavior that was easy to perform with Live Programming tools were frequently used. For instance, inspecting an object or executing an example are always one click away; so programmers used them very frequently. On the other hand, the Pharo search tools are fragmented: different tools perform different types of searches. The tools are both more difficult to use, and less frequently used. Knowledge is also a factor: accessing UI objects via halos was under-used, because many participants did not know the shortcut to activate it.

Few searches. Study participants used few searches, which contrast with studies that find code search to be a common behavior. Sadowski *et al.* [133] summarized these studies, finding code searching prevalent at Google. This finding agrees with Ko *et al.* [82], who found that developers preferred to use search tools that required less details. Another reason could be that users found other tools more effective, be it object inspectors, debuggers, or browsing source code references. Ko and Myers observed in an experiment that users of the Whyline (with extensive runtime information) relied much less on search than users of their control group [84].

Examples are more frequently used. Sillito observed that in only 26% of sessions (7 out of 27), participants searched for examples (asking question Q4). The use of examples occurred in 59% of our sessions (10 out of 17). Pharo has a more prevalent "culture" of examples, including executable method comments, classes and methods marked as examples, and dedicated example browsers. Our survey offers additional insights. We find agreement with Sillito when he states that finding examples is challenging, as our participants indicated that finding relevant examples was more difficult than executing or modifying them. We think that the ease of execution of examples is a significant factor. It is harder to execute, for instance, a Java code snippet found on Stack Overflow, as it needs to be imported (resolving dependencies) and compiled before one can try it. Approaches facilitating this process, such as Ponzanelli *et al.* [128] or Subramanian *et al.* [151] should significantly ease these issues.

Simple works best. Some of the most advanced approaches were under-used by our participants. While many debuggers were opened (every runtime error in Pharo opens one), they tended to be quickly dismissed; comparatively few of them were used in depth, or explicitly opened via breakpoints. While several respondents mention that they value being able to "program in the debugger" in our survey's free-form comments, we observed none in our sessions. Similarly, we previously mentioned that writing test cases and using halos were under-used: the former because it needs extensive knowledge of the code to test, and the latter for lack of awareness.

In contrast, the comparatively simpler tools such as the object inspector and the playground were used by virtually all the participants. These tools are easily accessible: any piece of code can be evaluated and inspected. They are also very versatile, and this is partly due to the possibility to open multiple instances of the tools at once, and keep them for a long period of time. Having several code snippets or object inspectors open at once makes it easy to perform comparisons between different scenarios. Keeping these tools open for long periods of time makes it easier to check one's progress. It is best exemplified by P7's use of two code snippets and two object inspectors simultaneously.

Comparison with online videos. While we observed some liveness in the analyzed online coding sessions, we found much less usage than for Pharo participants. All of the online coding sessions used the facilities afforded by immediate feedback (either by manual reload of applications, or automatically after source code changes) to check progress on their tasks. Nearly all (10 of 11) sessions using JavaScript and HTML made use of the object inspector provided by the web browser. However, the JavaScript inspector usage was more limited since the tool support is less mature. Other usages of liveness were minimal. This confirms our previous observation that tool support is vital for people to make use of liveness effectively.

4.8. Threats to Validity

Empirical studies have to make tradeoffs between internal and external validity [139]. As this study focuses on how developers used liveness in practice, we tried to maximize its external validity. To this aim, we performed our study in 3 steps.

We first observed 11 participants in 17 sessions who all used the same environment. To vary the situations, they worked on different tasks from ten different codebases. However, the duration of the tasks was short (40 minutes to an hour), which restricted the types of activities we observed. We could not do more for logistical reasons.

We further attempted to diversify our choice of settings by gathering 16 additional recorded coding sessions online, covering a variety of languages, settings, and tasks. We wanted to include sessions of the Jupyter interactive notebook [129], which has features similar to the playground. We were ultimately unsuccessful: we found mostly tutorials, not real-world coding sessions.

We had limited choice of participants, and our participants were all male; studies like the ones of Beckwith *et al.* [3] have shown that males are more comfortable than females with tinkering. A Live Programming environment does encourage tinkering. In a follow-up study of debugging by end-users [4], females were found to be less frequent, but more efficient users of tinkering, showing that the relationship is not at all obvious. As such, studies of how females use Live Programming would greatly extend this work.

The tasks we analyzed may not be representative or advantageous to a Live Programming environment, because such a representative task list does not exist yet. We only defined the tasks on unfamiliar code in the first study: we had no control on the tasks related to familiar code and the online video tasks, as both sets of tasks were defined by the participants themselves.

The transcription of the activity during the tasks was performed by the author of this thesis. In such cases where the author was unsure, we (the author and his supervisors) held discussions on specific examples. To minimize biases in interpretation, most of the activity under study was related to tool usage (plus vocalizations), and we used Minelli and Lanza's DFlow to systematically record interaction data [117] (except in one session).

According to McDirmid, Smalltalk environments such as Pharo do not cover the entire spectrum of Live Programming (a specific example is provided by McDirmid [111]). We nevertheless chose Pharo as it is the "liveliest" environment used by practitioners that we had access to, and note that Pharo does include Morphic [110], which fits McDirmid's stricter definition of Live Programming.

We kept the survey as short as possible to maximize responses. We had to make choices on what to ask, though we could not include every observation.

4.9. Summary and Implications

The live feedback that Live Programming provides is gaining acceptance in mainstream tools, *e.g.,* the React project from Facebook, the Swift language and the XCode development environment from Apple. We performed a multi-phase study on how developers use *liveness* in practice. After observing 17 development sessions of Pharo developers, we concluded that *they used liveness frequently*, and documented the various approaches they took. We found that participants favored *simple approaches*, and that liveness allowed them to *check their assumptions.* Advanced participants were able to easily compare and contrast execution scenarios; *practitioners may learn from the successful episodes we described.* A follow-up survey confirmed our observations on the frequency of usage of liveness. Finally, we contrasted our findings with online coding sessions in other languages, finding extensive use of immediate feedback, but limitations due to tool support.

Implications for tool builders and language designers. The major implication of this study is that *small doses of liveness* changes the way programmers explore and learn about software applications. Technologically impressive tools such as the Whyline [84] have been developed, while the Debugger Canvas [41] or TouchDevelop [12] have been put in production. These tools are extremely useful and required a lot of effort to build. Yet, we think that adding a playground and object inspector to an IDE could go a long way, particularly if multiple instances are allowed.

Similarly, Live Programming research focuses on languages with maximal liveness [112]. Yet we believe that exploring the design space of simpler representations such as the object inspector and playgrounds could lead to significant improvements on its own, particularly if longevity and multiplicity are included. The notebook metaphor employed by Jupyter [129] elegantly allows for multiple code snippets (cells) to coexist, while providing easy visualization of the results (particularly for data), even if it is not fully live. The moldable object inspector of Chis *et al.* [19] can be customized for each type of object to display the most relevant information. Combining both approaches is very intriguing as we document in this study.

Implications for software engineering researchers. The major implication is that there is still a lot to learn about how developers use liveness. As these features slowly but surely enter mainstream tools, multiple studies beyond this one will shed more light on how developers can best take advantage of the new capabilities their development environments will afford them.

Live Programming. In this and the previous chapters, we presented the study results on Live Programming. We first observed developer questions that Pharo users have during programming change tasks, considering the 44 questions by Sillito *et al.* as an initial list to this study. We shown that Pharo and other (surveyed) Smalltalk users prefer to understand programs by examining their behavior and structure at runtime. We can relate this observation to the previously reported one by LaToza *et al.* who conducted a survey and eleven interviews, concluding that "*developers avoid using explicit, written repositories of code-related knowledge in design documents or email when possible, preferring to explore the code directly and, when that fails, talk with their teammates*" [105].

The most commonly used tools and techniques in our study include writing and executing code snippets and inspecting objects. Pharo developers make extensive use of the ability to inspect the object states in object inspectors without being dependent on the debugger. We therefore observe that being able to evaluate code snippets and inspect applications (objects) at runtime are main Live Programming drivers. This goes in hand with the Live Programming common agreement: (i) Live Programming always offers accessible source code evaluation; and (ii) allows nearly instantaneous feedback to developers [123].

We have also shown that programmers ask questions that are still difficult to answer using Live Programming. Those questions typically involve many actions (sub-questions) to answer them. The current Live Programming thus does not lessen all the programming tasks and there is a room for the research and new solutions.

Chapter 5

Electronic Developer Communication

Whatever you repress and hide, it will only manifest later as destiny.

— Carl Gustav Jung

This chapter describes our motivation for choosing to focus on electronic developer communication within a Live Programming environments during programming change tasks.

5.1. Introduction

Electronic communication started in the 1960s when Ray Tomlinson invented electronic mail (email, or e-mail), that time a new method of exchanging messages among people using electronic devices. Some early email systems required the author and the recipient both be online at the same time. Today's emails systems are based on store-and-forward model in which neither users nor their computers are required to be online simultaneously. The email was standardized in 1973 which is a similar to a basic email sent today.

Email has also become a standard communication protocol for mass messaging, called electronic mailing lists. An author sends an email to a particular mailing list address and a special mailing list software distributes the received email to all email addresses subscribed to the mailing list. The first mailing list was created in 1971 and several mailing list solutions are used even today.

Nowadays alternative communication methods have emerged. It includes discussion forums, questions-and-answers (Q&A) sites, blogs, micro-blogs, and instant messaging. All of them use the store-and-forward model and users can receive and read messages at any time later, not necessarily at the moment of sending the message. Many communication solutions also integrate audio and video calls, which may or may not require conversation participants to be online at the same time.

In the previous chapters, we discussed how developers use Live Programming during programming change tasks. Given that most software is larger than a single programmer can comprehend on one's own, communication about the software among software developers is also of vital importance. Therefore, in this and following chapters, we discuss how Live Programming developers electronically communicate during programming change tasks.

We first define *live communication* as a concept of Live Programming in developer electronic communication. We define it using a term *levels of message liveness* similarly to Tanimoto's level of liveness defined in Chapter 2. We then explore research work related to software developer communication.

5.2. Live Communication

Tanimoto focuses solely on single developer situations and interaction of developers with their programming environments. But communication is a requirement on large-scale software development. What could it mean to communicate about Live Programming in a Live Programming environment? What is the fidelity of the information communicated to other developers? And how much help the programming environments give programmers in such electronic communication? Those are principal questions that drives our research.

We define four *levels of message liveness* as a level of fidelity of a communicated information. In this classification we solely focus on communicated information about source code and software application execution. For example, how to use an application, how to implement a feature, and how to fix an issue.

- *Level 1, Informative*: receivers are informed about a subject, but the information is not complete. Receivers must ask more questions or perform steps to understand it, *e.g.,* a message says that an application fails, but it is not obvious how to reproduce it; or a message suggests to use a particular API method, but extra code exploration is required to understand how to use it.
- *Level 2, Informative and significant*: a message includes all pertinent information about a subject. Receivers have all information at hand and can elaborate on the subject, *e.g.,* a message includes detailed reproduction steps; or a message includes an example of a particular API method usage.
- *Level 3, Informative, significant, and self-explanatory*: a message includes all pertinent information about a subject and also a content that explains effect of the discussed code, *e.g.,* a code that draws a blue rectangle is accompanied with a figure of the code output (rectangle).
- *Level 4, Informative, significant, self-explanatory, and live*: a message includes all pertinent information, is self-explanatory, and includes an *embedded* runtime environment (tool) where a shared code can be executed, edited, and explored.

We omit Tanimoto's level 5 and 6 as they are out of scope of the dissertation interest. Those levels involve artificial intelligence which is not discussed in this work. The predictions that are involved in those two levels should also include communicated message contents and help during the communication.

Each defined level comes up with a different communication experience and in the following text we provide communication examples for each defined level. We use a simplified discussion that is composed of one question and one answer. In the following examples, a *receiver* is a person who reads the question (in blue bubble).

Level 1, Informative. In this example, the receiver is informed about the subject, a method that he or she can use. He or she needs to perform extra steps to figure out how the method can be used. Those steps may include source code exploration, writing an example code to observe method call results.

How can I move an element to a specific position?

You can use `relocate:`

Level 2, Informative and significant. The following answer is more complete as the receiver can use the attached code snippet, run it and observe the execution results. The receiver is therefore free from having to explore source code and write an executable example code.

How can I move an element to a specific position?

You can use `relocate:` as follows:

```
parent := BlElement new.
parent background: (Color red alpha: 0.5).
parent size: 300@300.
parent clipChildren: false.

child := BlElement new.
child background: (Color blue alpha: 0.5).
child size: 300@300.
child relocate: 50@50.

parent addChild: child.
```

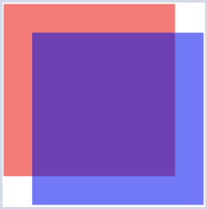
Level 3, Informative, significant, and self-explanatory. In addition to the previous level, the message also includes the code snippet result. The reader understands the code snippet meaning without having to read or execute the code snippet. It is particularly useful in this example, as the discussed method `relocate:` is accompanied with other eight code snippet lines.

In addition, the attached figure serves as a reference and the reader can compare his or her execution result with the attached one. Also a wider audience can follow the discussion as it does not require extra effort to comprehend it, considering that the communication happens in public.

How can I move an element to a specific position?

You can use `relocate:`, look at the code snippet and its execution result:

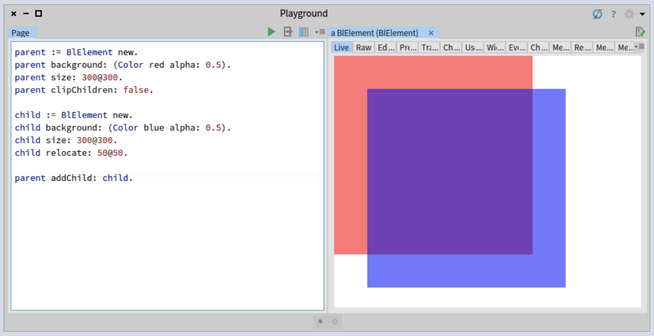
```
parent := BlElement new.
parent background: (Color red alpha: 0.5).
parent size: 300@300.
parent clipChildren: false.

child := BlElement new.
child background: (Color blue alpha: 0.5).
child size: 300@300.
child relocate: 50@50.

parent addChild: child.
```



Level 4, Informative, significant, self-explanatory, and live. The following answer includes an embedded playground that includes the accompanied code snippet and the execution result. The code snippet can be therefore edited and executed right at the message site. The execution result is no more *just* a figure, it is an inspected object and further object details can be explored right in the place.

How can I move an element to a specific position?

You can use `relocate:` as you can see in the playground:



```
parent := BlElement new.
parent background: (Color red alpha: 0.5).
parent size: 300@300.
parent clipChildren: false.

child := BlElement new.
child background: (Color blue alpha: 0.5).
child size: 300@300.
child relocate: 50@50.

parent addChild: child.
```

Such a solution opens new questions, for example, whether the embedded playground content changes are or can shared instantly among message readers. For the purpose of this dissertation, all embedded development tools are local and code and navigation changes are not instantly shared. If a user wants to share (for example) a changed code snippet, he or she sends it as a new message.

In the rest of the chapter, we present different communication media (Section 5.3), and other coordination solutions (Section 5.4) that are used today.

5.3. Communication Media

Existing solutions employ different media, for example mailing lists, blogs, micro-blogs, social networking, Q&A sites, instant messaging. Storey *et al.* provide a broad review of different communication channels in software engineering since 1968 [148]. In their study, developers mentioned face-to-face communication as important, followed by Q&A community sites, micro-blogs, private instant messaging, and blogs. In the remainder of the section we mention the most prominent communication mediums in the present day.

Emails. Email was standardized in 1973 and is widely used today by millions of people. Whittaker *et al.* summarize its usage and advantages and disadvantages [165]. It is prominent for two reasons: both senders and recipients can communicate at times and in places that are convenient for each other; and it is malleable. People use it for communicating, sharing documentation, archiving, keeping contacts, managing tasks, etc.

Although email is used in various scenarios, standard applications do not support those scenarios effectively. Whittaker *et al.* surveyed empirical studies on email usage and propose improvements for distinct email usages. They conclude that mobility, text processing, and task management has to be improved.

Tenório and Pernille also point out some issues using emails. In particular, they report that navigating substantial email conversations is problematic. It is often challenging to comprehend and follow different interaction lines (emails) fully. Users of their study also highlight that the formal language in emails hinders an effective communication. Users find informal conversations through chats more effective [157].

Bacchelli *et al.* improve developer email communication experience by introducing Remail that integrates the email protocol with the Eclipse development environment and links discussions with source code [1]. Remail displays relevant email messages next to an observed source code, supporting four program comprehension scenarios: (i) it helps find entry points (related to the Finding Focus Points questions at Section 3.2.1) based on most discussed source code; (ii) it assists during software evolution analysis by providing relevant historical email discussions; (iii) it suggests experts about specific code entities; and (iv) it recovers additional information that is not available in source code comments and documentations. The way Remail links messages and source code is presented in their previous work, where they benchmark various solutions [2].

However, another later work by Guzzi *et al.* shows that development mailing lists are no longer the key communication channel and discussion is scattered among repositories. They acquired this knowledge by analyzing mailing lists of open source software development teams [62].

Q&A sites. Q&A stands for *questions and answer* and Q&A sites are online services that attempts to answer questions. They integrate users in similar fields to discuss questions and provide answers. Q&A sites emerged in the late 1990s, when a free online service called "Answer Point" was launched [36]. Since then, more and more sites were available: Google Questions and Answers in 2001, Ask Yahoo! in early 2000s, and Q&A Stack Overflow in

2008.

The last mentioned, Stack Overflow [72], is a popular Q&A site for software engineers, focusing on computer programming. Q&A services archive millions of entries that contribute to the knowledge body in software development, being often a substitute of the official documentation [159].

Wu *et al.* analyzed software projects that contain explicit references to Q&A Stack Overflow posts. They were interested in barriers that hinder the code reuse and founded that developers had to modify the Q&A Stack Overflow code in 31.5% cases. Changes varied from variable renaming to rewriting the whole algorithm. In a subsequent survey, they found that the top three barriers are: (i) too many necessary code changes to fit in their projects, (ii) incomprehensible code, and (iii) low code quality. Developers suggest an integrated validation mechanism to keep the code of a high quality and to detect outdated code [167].

Ponzanelli *et al.* argue that having Q&A services isolated from the development environments hinders the benefits brought by the services [127]. They therefore introduce Seahawk, an Eclipse plugin, to assist programmers using Stack Overflow. Seahawk automatically formulates queries based on a developer active content and suggest relevant Stack Overflow discussions. In addition, it lets users import code samples.

Rebecca Tiarks analyzes content structure of Stack Overflow [158], *e.g.,* structure, length, topics. She concludes that the site contains information that is rarely found in tutorials, the posts are mostly unstructured, and nearly half of them include source code.

Ponzanelli *et al.* use this fact and introduces Prompter, an Eclipse plug-in [128]. The solution automatically retrieves pertinent discussions from the Q&A Stack Overflow website and brings the content to the development environment. They report that the Prompter recommendations were positively evaluated and significantly help developers to improve tasks correctness. They also observe that 78% of the Prompter recommendations are volatile and may change within a year.

Blogs and micro-blogs. Wikipedia defines a blog as *"a discussion or informational site published on the World Wide Web consisting of discrete entries (posts) typically displayed in reverse chronological order (the most recent post appears first)"* [30]. It is a medium for publishing daily journals, expressing ideas, and sharing knowledge.

The modern blog evolved in 1994. At this time Justin Hall began a personal blog and since then he is generally recognized as one of the earliest bloggers [34]. The term blog was coined by Peter Merholz in 1999 [30]. Shortly after the term blog was used as both a noun and verb. The first micro-blog appeared in 2005 and become widely used when Twitter and Tumblr were founded in 2006 and 2007.

As blogs and micro-blogs become popular in software engineers, Parnin *et al.* interviewed developers and analyzed their blog posts to find out why they put an effort into writing blog posts about their knowledge, experiences, and opinions [124]. They observe that documentation, including tutorials and technology experience reports, are the most frequent posts, followed by progress reports and technology discussions, among others. Developers are motivated to publish for diverse reasons such as personal branding, knowledge retention, and feedback.

Efimova and Grudin present a study of blogging practices in the Microsoft corporate setting. Employees initiate personal blogs as it is suggested by other prominent persons and blogging proves to be a good communication tool with others [48]. Johri evaluated blogging software usage in a global virtual company and reports that use of blogs was effective in sharing understanding, integration, and trust [75, 76].

Micro-blog differs from a traditional blog in that its content is typically shorter; Twitter [73] is a worldwide example. Singer *et al.* asked developers how they use Twitter and report that they use it to stay aware of industry changes, for learning, and for building relationships [144]. Begel *et al.* report that engineers use micro-blogging to publish links to technical information, to announce spontaneous meetings, and to inform co-workers about progress [5]. Díaz and Arellano introduce an micro-blogging integration into domain specific language editors [44] and they observed that users tend to be inaccurate expressing what they want, potentially leading to contributors having doubts about what is expected.

Guzzi *et al.* integrate micro-blog messages into the Eclipse development environment, encouraging users to share short messages about their current task and progress, *e.g.,* posting messages as "*trying to figure out how to create a proper UUID from an int in the database.*" They report that developers are willing to share short messages and shared messages expressing future intentions and status updates on concluded and ongoing activities. More than 25% of all collected messages included explicit references to a code such as class names. Their exploratory study indicates benefits, integrating micro-blogging within development environments [63].

Instant messages. Instant messaging is a type of online chat technology that offers real-time message transmission over the Internet. It actually predates the Internet as first solutions appeared in the mid-1960s. They were initially used as notification systems for services like printing, but quickly were used for communication with other users logged into the same machine [35].

Over time, IRC, Skype, Slack, and Facebook Messenger, among many others, were developed and represent a type of online chat that offers real-time communication today. They are not limited to real-time chat, neither to one machine, and senders and recipients can communicate at times that are mutually convenient. Researchers point out benefits of using instant messaging in companies and in educational institutions.

Shihab *et al.* analyzed discussions of an open source community using Internet Relay Chat (IRC); they found that IRC gains popularity among open source developers and maintainers [138]. Johri presents a company use case, where IRC and blogging are the primary communication tools, improving the firm coordination and knowledge sharing [75]. Lin *et al.* found that Slack, a communication platform for teamwork, and its integrations (bots) play significant role in software development [106]. They found that Slack supports team collaboration through team management, file and code sharing, DevOps notifications and software deployments.

Tenório and Pernille explored how the chat technology allowed a distributed software team to successfully coordinate software development. The team included both vendor and client team members. They observed that textural and permanent nature of the chat was crucial in the communication. Prior conversations supported reflective behavior and potential re-submission of past conversations. Also synchronous chat interaction supported the collaboration, unlike emails, where following different interaction lines is challenging [157].

Hundhausen and Carter introduce a social programming tool in an educational programming environment, promoting social interaction and awareness. Users of their OSBIDE (Online Studio-Based IDE) Microsoft Visual Studio plugin, can read activity streams, including compilation issues, run-time exceptions, and debugging events. Their plugin then relates user activities to other learner activities, displaying messages as "*You and 3 others have gotten this error*" [68].

5.4. Real-time Collaboration

To encompass other collaboration techniques, we present real-time collaboration options, imitating co-located pair-programming in dislocated setups. Those solutions can be used in conditions where several dislocated co-workers work on the same project, requiring having the latest changes from all collaborators all the time.

Remote Pair Programming. Collaboration mode called *Remote pair programming (RPP)* is usually implemented by screen sharing [136]. One pair member desktop is transferred to the partner's computer via a VNC, Skype or many other solutions. It provides WYSIWIS (What You See Is What I See) setup; both developers see the same editor context and editing activities [147].

RPP involves two asymmetric situations: The remote partner suffers from twice the network latency when editing; and has to deal with the partner's IDE configuration, *e.g.,*. colors, layout, key bindings [136].

Distributed Pair Programming. Distributed pair programming (DPP) removes the RPP asymmetry. Development tools instantly replicate file changes and make all editing operations local; and both developers use their preferred IDE configuration.

Sangam is an Eclipse plug-in that allows dislocated developers to share same programming environment as they were using the same computer [67]. While collaborating remotely, one developer is a driver who has a control over a mouse and a keyboard. Developers can switch roles at any time. XPairtise is another Eclipse plug-in that improves pair programming in a distributed environment setting. Users also adopt driver and navigator roles, defining who controls a mouse and a keyboard. In addition to editor and audio sharing, developers can also exchange ideas using a shared graphical whiteboard and a text chat. Schümmer and Lukosch evaluated the tool and report that role switches did not occur often and the most communication happened via the audio channel [137].

There are other solutions that allow editing freedom, *e.g.,* VS Anywhere, Saros, and Collabode. Developers can switch to a mode in which one partner follows the activity of the other partner. The mode permits conversation with partners whenever a developer does not know how to deal with a situation. Once it is clear, programmers can continue working independently, while their source code is continuously synchronized.

VS Anywhere is a fully-integrated Visual Studio extension that activates a real-time multi-person collaboration [70]. Saros is an Eclipse plug-in supporting a distributed party programming with two or more developers working together [134]. Schenk *et al.* evaluated how developers can handle such programming environment and observed a pair of industrial software programmers for three days [136]. They conclude that the distributed-pair programming can work as well as local pair programming, if applied skillfully in an appropriate context.

Goldman *et al.* describes Collabode, a web-based Java integrated development environment [60]. Collabode is designed to support close and synchronous collaboration between developers, involving an error-mediated integration of program code to handle compilation errors introduced by other co-workers. Goldman *et al.* conclude that the solution provides benefits over a naive continuous code synchronization.

5.5. Summary

In this chapter we defined level of message liveness for the context of communicating Live Programming in a Live Programming environment. The individual levels are accompanied by an illustrative example. Each example shows that the higher the level number, the higher the message fidelity is. Users thus comprehend shared information *easier*.

The easier comprehension means that users gain understanding about a discussed subject faster. But time is not only variable: easier comprehension also means that message readers are more confident about the way they understand the shared information. The shared information also can be understood in more detail, if this is a subject of user interest.

Levels of message liveness are equivalent to Tanimoto's levels of liveness. In both cases, users comprehend information more easily. In the case of level of liveness, it may be a fact that users observe and comprehend an effect of code changes faster, excluding manual compile and re-execute cycles. In the case of level of message liveness, it may be a fact that users do not need to perform extra source code exploration or execution to understand a discussed subject.

The discussion about communication media and remote collaboration techniques shows that developers use different communication solutions depending on personal preferences and project and company policies. Researches therefore provide solutions that simplify information sharing in different communication medias. The provided solutions have in common that they enrich shared content with extra context specific information, and brings relevant external information into software development environments.

In this dissertation, we explore how developers discuss Live Programming in Live Programming environments. For this purpose we implemented LightShare, a chat communication tool integrated in a Live Programming environment that simplifies sharing information. The solution is inspired by the observed fact, that developers are willing to share their task progress, gained knowledge, and do it more likely with an appropriate tool support. We contribute to the research field by exploring what the communication is about when users discuss Live Programming.

Chapter 6

Sending Software Artifacts From the IDE to the Chat with LightShare

Problems cannot be solved with the same mindset that created them.

— Albert Einstein

In the previous chapters, we report that Live Programming eases program comprehension and Pharo developers favor dynamic information and feedback over static information. In the following research, presented in this chapter, we extend our liveness interest to the developer electronic communication during a change task. We propose LightShare, a tool to simplify communication within Pharo, enriching chat messages with additional information and software artifacts. We first explain how LightShare works, and than we provide its evaluation.

6.1. Introduction

Software development is often a team effort where professionals collaborate to produce and maintain software systems [164]. Knowledge management plays an important role in the process and can be recorded and distributed using a variety of media, including help files, manuals, videos, technical articles, wikis, and blogs [160]. Nevertheless, the current support for reusing and sharing program comprehension knowledge is still limited and researchers look for solutions in adopting social media during development process [148]. For example, Guzzi *et al.* encourage developers to share their knowledge in Twitter-like micro-blog messages [63] and Singer *et al.* surveyed and interviewed developers active on GitHub how Twitter helps them to keep up with industry changes [144].

In this chapter, we explore how to better support software developer communication when a developer struggles in his or her task and requests help from co-workers or to a community. In particular, we focus on the problem of sharing software artifacts. LightShare, our lightweight IDE integration, is inspired from a common communication integration, predominately observed in mobile systems: Users initiate sharing an artifact from nearly any application and then continue a discussion in the particular chat service, e.g., WhatsApp, iMessages, Facebook Messenger. Similarly, LightShare simplifies sharing software artifacts within a development environment, with the intention that the discussion continues in the particular chat service where the artifact was shared.

We opt for this *light* proposal for one main reason: if developers find this workflow (initiating discussion in development environments, switching to dedicated chat clients), we can propose a unified interface and integrate other chat and publishing services, *e.g.,* Twitter (as it was proposed by a participant in this study), without dealing with full client support for each service. Similarly, we could adopt the experience of the previous research by Guzzi *et al.* [63] and integrate micro-blogging services.

LightShare is developed by the dissertation author and is implemented for the Pharo development environment [7] and the Discord service [45]. We chose Pharo in order to research how Pharo developers communicate and what software artifacts they (want to) share while using the Live Programming environment. Although LightShare is mature enough for an initial deployment, we consider the evaluation as a preliminary (a pilot study) with the main purpose to receive a feedback from its users.

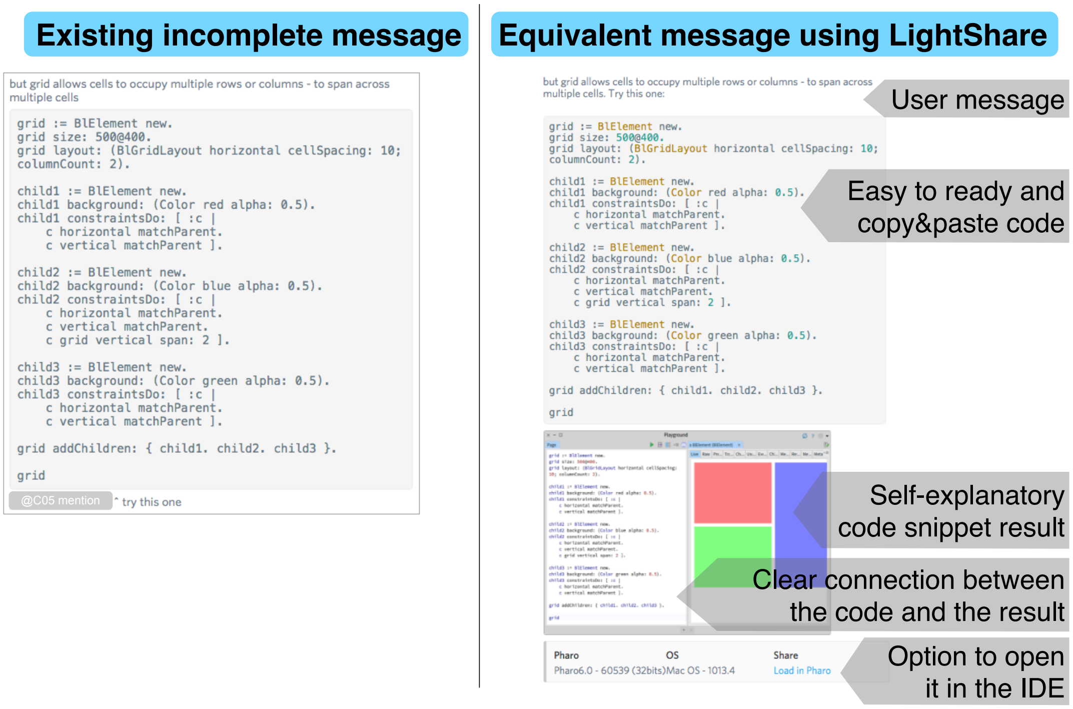
Research Objectives. The aim of our research effort is to gather insight on the effects of a lightweight developer communication approach integrated in an IDE. To address it, we set the following objectives to this study:

- Evaluate how developers use LightShare:
  - *How often was LightShare used during the study?* We find that LightShare was used relatively infrequently.
  - *What information is shared using LightShare?* The most common shared artifacts were code snippets, screen captures, and combinations thereof; animated GIFs and source code packages were rarely shared.
  - *What are the scenarios where LightShare was involved?* LightShare was involved in a variety of scenarios, including increasing readability, learning tools, helping beginners, and reporting bugs.
- Evaluate user satisfaction using LightShare:
  - *Are developers satisfied with LightShare?* Participants agreed that LightShare is useful, particularly to share screen captures.
  - *What do they propose to improve?* Participants point out a variety of possible improvements.

Structure of the chapter. We start by presenting LightShare, alongside a motivating example (Section 6.2). We then present the methodology of the study (Section 6.3), followed by results (Section 6.4). Finally, we discuss the feedback we obtained from the participants (Section 6.5) before concluding (Section 6.7).

6.2. LightShare

In this section, we provide a motivating example, describing LightShare objectives and strengths. We then describe how LightShare works, followed by our initial scope, design decisions, and current tool limitations.


Existing incomplete message

but grid allows cells to occupy multiple rows or columns - to span across multiple cells

```
grid := BlElement new.
grid size: 500@400.
grid layout: (BlGridLayout horizontal cellSpacing: 10;
columnCount: 2).

child1 := BlElement new.
child1 background: (Color red alpha: 0.5).
child1 constraintsDo: [ :c |
    c horizontal matchParent.
    c vertical matchParent ].

child2 := BlElement new.
child2 background: (Color blue alpha: 0.5).
child2 constraintsDo: [ :c |
    c horizontal matchParent.
    c vertical matchParent.
    c grid vertical span: 2 ].

child3 := BlElement new.
child3 background: (Color green alpha: 0.5).
child3 constraintsDo: [ :c |
    c horizontal matchParent.
    c vertical matchParent ].

grid addChildren: { child1. child2. child3 }.

grid
```

@C05 mention ^ try this one

Equivalent message using LightShare

but grid allows cells to occupy multiple rows or columns - to span across multiple cells. Try this one:

```
grid := BlElement new.
grid size: 500@400.
grid layout: (BlGridLayout horizontal cellSpacing: 10;
columnCount: 2).

child1 := BlElement new.
child1 background: (Color red alpha: 0.5).
child1 constraintsDo: [ :c |
    c horizontal matchParent.
    c vertical matchParent ].

child2 := BlElement new.
child2 background: (Color blue alpha: 0.5).
child2 constraintsDo: [ :c |
    c horizontal matchParent.
    c vertical matchParent.
    c grid vertical span: 2 ].

child3 := BlElement new.
child3 background: (Color green alpha: 0.5).
child3 constraintsDo: [ :c |
    c horizontal matchParent.
    c vertical matchParent ].

grid addChildren: { child1. child2. child3 }.

grid
```

Pharo	OS	Share
Pharo6.0 - 60539 (32bits)	Mac OS - 1013.4	Load in Pharo

Playground
Page
User message
Easy to ready and copy&paste code
Self-explanatory code snippet result
Clear connection between the code and the result
Option to open it in the IDE

Figure 6.1: Left: a message with a code snippet explaining how a graphical layout works. Right: the equivalent message with LightShare. The message includes additional information to easily follow the conversation.

6.2.1. Motivating Example

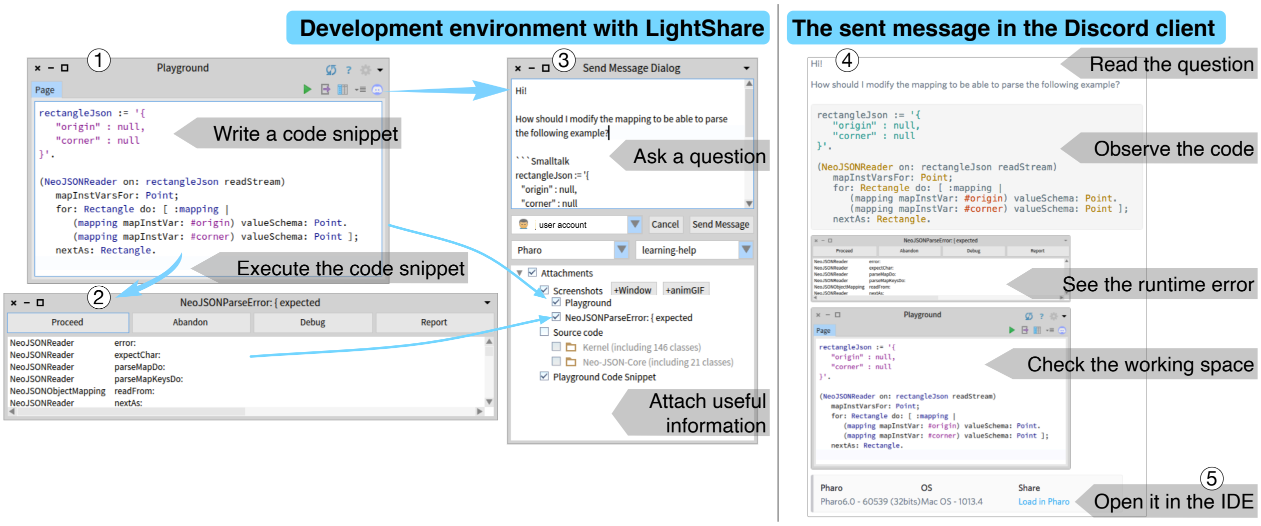
LightShare's goal is to ease developer communication by simplifying the sharing of the software artifacts that they wish to discuss. To illustrate LightShare, we present a discussion extracted from a public chat we analyzed in this study, from a conversation that happened before our study execution. Users C05 and C16 discussed how to implement a certain graphical layout.

- C05 wrote "*[. . .] How [does one] do [. . . a] table layout with wrapping? [. . .] But grid requires concrete columns count[, right]? [. . .]*"
- C16 followed "*yes, it does not auto wrap [. . .] but grid allows cells to occupy multiple rows or columns - to span across multiple cells.*"
- C16 then sent a code snippet with about 20 lines of code to demonstrate how the grid layout works, and continued "*[. . .] try this one.*"

The left side of Figure 6.1 includes part of the discussion with the code snippet. Readers who want to follow the discussion have to take the code snippet and execute it in order to see its result. C05, who was actively involved in the discussion, evaluated the code snippet and said "*Yes, I saw it.*" However, C16 could post an easier-to-follow message, by directly including a code snippet result; this would allow additional readers, perhaps unwilling to spend the effort to execute the code snippet, to also see the result. This would also allow C05 to check whether the observed code snippet result is indeed the same as C16's.

The equivalent LightShare message is presented on the right side of Figure 6.1. Any reader, not necessarily actively involved in the discussion, could capture the meaning of the code snippet without leaving the chat. In addition, the code might not work anymore as source code evolves and it might be hard to understand the received information. LightShare also provides a link to instructions that allow them to load the code snippet and see its result directly in the IDE, should they wish to do so.

6.2.2. How LightShare Works



```
rectangleJson := '{
   "origin" : null,
   "corner" : null
}'.

(NeoJSONReader on: rectangleJson readStream)
   mapInstVarsFor: Point;
   for: Rectangle do: [ :mapping |
      (mapping mapInstVar: #origin) valueSchema: Point.
      (mapping mapInstVar: #corner) valueSchema: Point ];
   nextAs: Rectangle.
```

NeoJSONReader	error:
NeoJSONReader	expectChar:
NeoJSONReader	parseMapDo:
NeoJSONReader	parseMapKeysDo:
NeoJSONObjectMapping	readFrom:
NeoJSONReader	nextAs:

Hi!

How should I modify the mapping to be able to parse the following example?

~~~
```Smalltalk
rectangleJson := '{
  "origin" : null,
  "corner" : null
~~~

▾ ☑ Attachments
- ☑ Screenshots +Window +animGIF
  - ☑ Playground
  - ☑ NeoJSONParseError: { expected
- □ Source code
  - □ Kernel (including 146 classes)
  - □ Neo-JSON-Core (including 21 classes)
- ☑ Playground Code Snippet

Hi!

How should I modify the mapping to be able to parse the following example?

```
rectangleJson := '{
   "origin" : null,
   "corner" : null
}'.

(NeoJSONReader on: rectangleJson readStream)
   mapInstVarsFor: Point;
   for: Rectangle do: [ :mapping |
      (mapping mapInstVar: #origin) valueSchema: Point.
      (mapping mapInstVar: #corner) valueSchema: Point ];
   nextAs: Rectangle.
```

Pharo	OS	Share
Pharo6.0 - 60539 (32bits)	Mac OS - 1013.4	Load in Pharo

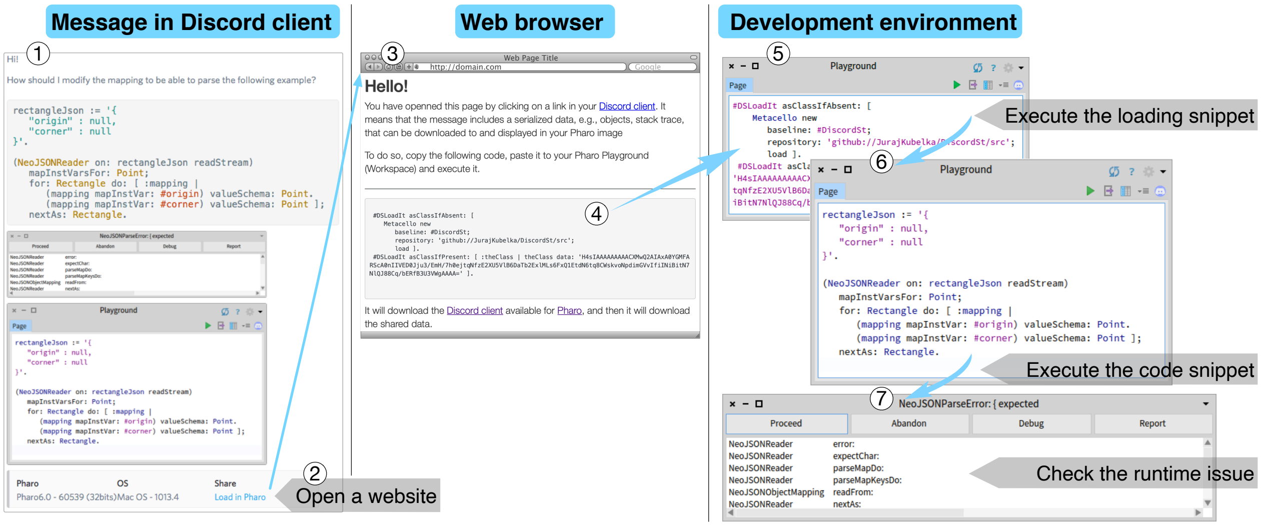
Figure 6.2: Left: How developers can ask a question from the IDE with LightShare, including software artifacts. Right: What the final message looks like in a Discord chat client.

Sharing artifacts. Figure 6.2 shows how LightShare is integrated in the IDE. In this case, a user has a partially working code snippet (part 1) that results in an error (part 2). He or she decides to ask for help in a chat, clicks on a playground's button that opens a LightShare dialog (part 3). He or she writes a question which automatically includes the code snippet and the playground screenshot. He or she includes the debugger window screenshot, and sends the message. The right side of the figure shows the format of the message in the chat client LightShare is integrated with (part 4); currently, LightShare supports the chat platform used by the Pharo community, Discord. Readers can read the question, observe the code snippet, follow the error message and screenshot of the working space.

LightShare also facilitates sharing source code. LightShare detects the source code dependencies (considering class references) that are necessary to run the code snippet and shares them alongside the code snippet. This is useful in situations where users have code that is not published yet, but need help nevertheless. Readers thus do not need to load the extra source code from external sources, *e.g.*, GitHub, and can run the code easily. Note that this feature is meant to be used to ease the sharing of simple use cases (e.g. the size of a code snippet); it does *not* suplement sophisticated source code repositories such as GitHub.

LightShare also simplifies sharing animations in the form of animated GIF files. Users can define a recording area and take necessary steps to expose information they want to share. Users can use it to share graphical animations, application and tool interactions, and steps to reproduce bugs.

Loading artifacts. Figure 6.3 shows how readers, willing to work with a shared artefact (part 1) on Discord, can open the task in their Pharo development environment. A reader first clicks on the "Load in Pharo" link (part 2). The link redirects the reader to a web page (part 3), explaining the context and how to open the shared artifact in Pharo. The web page includes a loading script (part 4) that the reader is asked to copy and paste to a Pharo Playground (part 5) and then execute. The loading script loads LightShare if it is missing in the running Pharo, followed by fetching and extracting of the shared artifacts (part 6). In the particular example, the previously published Playground content is opened (compare Figure 6.2 part 1 with Figure 6.3 part 6). The reader can execute it and reproduce the issue (part 7). If the reader is willing to help, he or she can fix the issue and share a solution as described in the previous paragraph "Sharing artifacts".



Hi!

How should I modify the mapping to be able to parse the following example?

```
rectangleJson := '{
   "origin" : null,
   "corner" : null
}'.

(NeoJSONReader on: rectangleJson readStream)
   mapInstVarsFor: Point;
   for: Rectangle do: [ :mapping |
      (mapping mapInstVar: #origin) valueSchema: Point.
      (mapping mapInstVar: #corner) valueSchema: Point ];
   nextAs: Rectangle.
```

```
NeoJSONReader          error:
NeoJSONReader          expectChar:
NeoJSONReader          parseMapDo:
NeoJSONReader          parseMapKeysDo:
NeoJSONObjectMapping   readFrom:
NeoJSONReader          nextAs:
```

Pharo	OS	Share
Pharo6.0 - 60539 (32bits)	Mac OS - 1013.4	Load in Pharo

Hello!

You have openned this page by clicking on a link in your Discord client. It means that the message includes a serialized data, e.g., objects, stack trace, that can be downloaded to and displayed in your Pharo image

To do so, copy the following code, paste it to your Pharo Playground (Workspace) and execute it.

```
 #DSLoadIt asClassIfAbsent: [
    Metacello new
       baseline: #DiscordSt;
       repository: 'github://JurajKubelka/DiscordSt/src';
       load ].
 #DSLoadIt asClassIfPresent: [ :theClass | theClass data: 'H4sIAAAAAAAACXMwQ2AIAxA0YGMFA
RScA0nIIVED0Jju3/EmH/7h0ejtqNfzE2XU5VlB6DaTb2ExlMLs6FxQ1EtdN6tq8CWskvoNpdimGVvIfiINiBitN7
NlQJ88Cq/bERfB3U3VWgAAAA=' ].
```

It will download the Discord client available for Pharo, and then it will download the shared data.

Figure 6.3: Left: message shared using LightShare with an option to load a shared artifact to Pharo IDE. Middle: a web page explaining how to load the shared artifact into Pharo. Right: the artifact with reproduced issue in Pharo

6.2.3. LightShare Pre-Study Survey

Survey. To get a first understanding of how Pharo developers communicate, the survey presented in Chapter 4 includes questions about communication practices. The first question is "*How often do you ask for examples, program usages, and/or issue solutions: to (i) colleagues in person; (ii) in chats and/or mailing lists?*" with the possible answers: (a) rarely, (b) occasionally or sometimes, (c) regularly, (d) often, (e) very often, (f) I do not know, and (g) not applicable.

The second question is "*What do you use (attach) when discussing programming issues electronically?*" and we asked explicitly for the following software artifacts: (i) screenshots, (ii) code snippets, (iii) source code, (iv) links to source code, (v) patches or bug fixes, (vi) stack traces in text forms, (vii) serialized objects and/or programs, and (viii) serialized stack traces. Participants were able to answer: (a) I do it easily; (b) I do, but exchanging it is difficult; (c) I do not, because exchanging it is too complex; (d) I would like to, but I am not authorized to do this; (e) I do not want or need to; (f) I do not know; and (g) not applicable. In addition, participants were allowed to write additional free-form replies.

Responses. Figure 6.4 shows how often practitioners ask for source code examples, program usages, and issue solutions to colleagues in person (face-to-face). Out of 190 respondents, 58 ask regularly or (very) often. Out of 190 respondents, 109 ask rarely or occasionally in person. Less respondents, 42 out of 190, ask regularly or (very) often in chats and mailing lists. Out of 190 respondents, 139 ask rarely or occasionally in chat and mailing list channels.

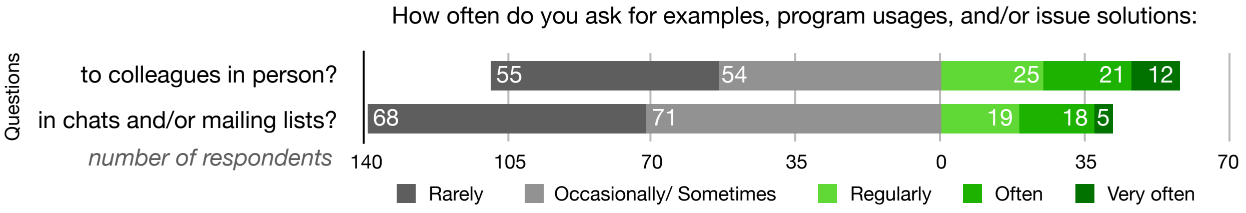


Figure 6.4: Surveyed participant answers on how often they communicate with their colleagues.

Figure 6.5 uncovers what artifacts practitioners share while discussing programming issues electronically. We observe that code snippets are favored (159 out of 190 respondents), followed by source code (123 out of 190) and screenshots (102 out of 190). A couple of respondents answered that sharing screenshots and source code is complex, while none of the respondents answered so in the case of code snippets.

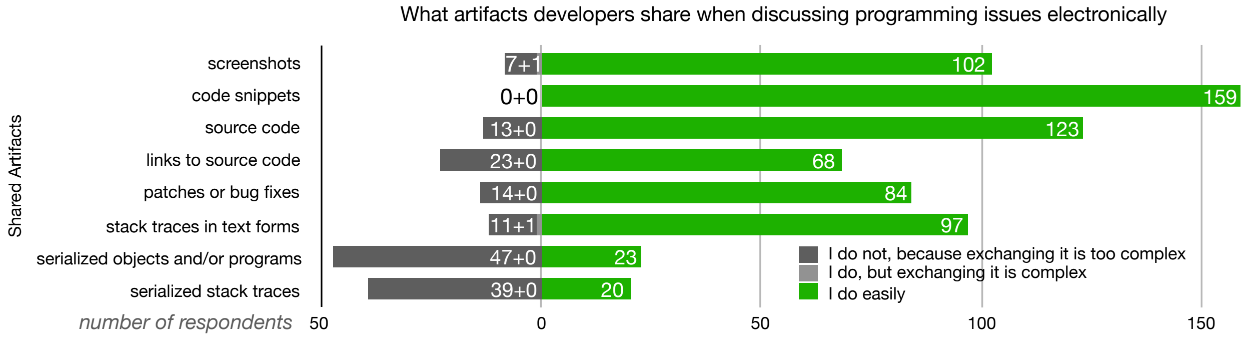


Figure 6.5: Surveyed participant answers on what software artifacts they share during electronic communication with their colleagues.

Initial scope. The evaluated LightShare implementation, presented in Section 6.2.2, integrates sharing screenshots, code snippets, and source code (which may also represents patches and bug fixes). As our respondents answered that sharing serialized objects and stack traces is difficult, our LightShare prototype also includes those options. Users are able to serialize objects and send them as a message to Discord. Receivers are then able to load the message and *materialize* (de-serialize) objects in their programming environments. We involve a FUEL technology presented by Dias *et al.* [43] for the object serialization and materialization process.

By the time we were about to conduct the evaluation study, we had some technical issues, that forced us to disable object and stack trace sharing for the study. Specifically, we did *not* resolve situations when recipients have a different source code version of the shared objects. The FUEL technology detects such situations, handling and exchanging source code in such circumstances is however missing in the existing LightShare version.

Partial vs. full integration. LightShare is a *partial* chat integration that foster message sending with additional context information we describe in the Subsection 6.2.2. Developers read responses and continue discussions in a standard chat client application (not integrated to Pharo). A *full* chat integration, which we did not implement, would then allow all chat communications to be carried out in Pharo.

We went for the partial chat integration for two main reasons. First, it requires less effort to implement it. If such a solution is accepted, it will be easier to adopt a variety of other services to integrate, *e.g.,* Slack, Twitter, GitHub. Second, the prior study by Guzzi *et al.* who integrated micro-blogging [63], provides strong indication that the *light* (partial) solution has great potential. They also report that "*developers are willing and inclined to share what they are doing by means of a short micro-blog message, regardless from the setting in which they work.*"

6.3. Research Method

To measure the impact of LightShare on practitioners, we designed a methodology to monitor the usage of our prototype. Table 6.1 shows the overall study course.

Date	Description
Feb 23, 2018	Contacting the 35 most active Pharo users on the public Discord channels related to Pharo
March 12, 2018	Start of the five-weeks long daily study
March 12, 2018	Initial survey
March 19, 2018	Weekly survey number 1
March 26, 2018	Weekly survey number 2
April 2, 2018	Weekly survey number 3
April 9, 2018	Weekly survey number 4
April 15, 2018	End of the study
April 16, 2018	Final survey

Table 6.1: Study timeline summary.

In this section we describe the study setup and tools we used to obtain data. In Section 6.3.1 we explain why we did this particular type of evaluation. In Section 6.3.2 we describe the case study implementation and the preparatory considerations. In Section 6.3.3 we detail tools we used and our rationale for doing so. In Section 6.3.4 we discuss the study threats to validity.

6.3.1. Study Decisions

We decided to conduct the daily study in a real environment because we wanted to clarify the tool applicability in real conditions. As a result, we received user responses that would be difficult to obtain in another study setup, *e.g.,* in a controlled experiment. For example, the reluctance to download shared source code to participant development environments (computers). We would also hardly obtain different user preferences on degree of chat integration.

Considering another study setup, it would be also challenging to provide an objective collection of tasks that would *naturally* lead to a chat communication with other persons. And it would be even more challenging to objectively instruct people assisting during the chat discussion.

6.3.2. Study Design

Participants. We contacted the 35 most active Pharo users on the public Discord channels related to Pharo. They were most active during a period of three months before the study (from November 2017 to January 2018). Participants were contacted two weeks in advance. From the 35 contacted users, 22 accepted to participate in the study, see Table 6.2. Participant programming experience range from less than 2 years to more than 10 years. Ten participants received their experience in industry (7 out of 10 with positions in industry and academia), 6 in academia (1 also as a hobby), and 6 in hobby projects. Ten participants reported that they program on daily basis (mainly people involved in industry), 5 program every week, 7 less then weekly (mainly hobby projects).

Out of 22 people who accepted to participate in the study, 15 participants ended up using LightShare during the study period. They are marked with the * character in Table 6.2. In addition to the explicitly invited participants, six additional users installed LightShare during the study period, presumably as a consequence of seeing its usage on public Pharo channels. We use letter C (as contributor) referring to explicitly invited participants. Letter U refers to the six additional users who used LightShare during the study. The additional users did not participate in our regular questionnaires.

Study Setup. On the first day of the study, we sent a message to all the participants, informing them about the study, with links to the LightShare installation and usage documentation, and a link to an initial, mainly demographical survey. During the first week of the study, 22 participants filled the initial survey.

At the beginning of the each week, we sent them a message with a link to a weekly survey where they filled their experience and developer behavior of the past week. We also used a dedicated Discord channel where the tool users shared their thoughts, issues and recommendations. Both means of communication helped us keep participants using the tool,

Participant (Contributor)	Experience [years]	Where	Frequency	Room
C01	6–10	Hobby	Less than weekly	Alone
C02*	3–5	Academia	Daily	Alone
C03*	3–5	Industry	Daily	Alone
C04*	3–5	Industry, Academia	Daily	Shared office
C05*	> 10	Industry, Academia	Daily	Shared office
C06*	> 10	Industry	Daily	Shared office
C07	≤ 2	Academia, Hobby	Weekly	Remotely
C08*	3–5	Industry, Academia	Daily	Shared office Remotely
C09*	> 10	Industry	Weekly	Remotely
C10*	6–10	Academia	Weekly	Single office
C11*	≤ 2	Hobby	Weekly	Alone
C12	≤ 2	Academia	Less than weekly	Remotely
C13*	6–10	Hobby	Weekly	Alone
C14	≤ 2	Academia	Less than weekly	Remotely
C15*	6–10	Industry, Academia	Daily	Single office
C16*	3–5	Industry, Academia	Daily	Remotely
C17	≤ 2	Academia	Less than weekly	Alone
C18*	3–5	Industry, Academia	Daily	Single office
C19	≤ 2	Hobby	Less than weekly	Alone
C20	≤ 2	Hobby	Less than weekly	Alone
C21*	> 10	Industry, Academia	Daily	Shared office
C22*	≤ 2	Hobby	Less than weekly	Alone

Table 6.2: Participants, their programming experience, habits, and environment.

as a possible loss of participants was the greatest danger of the study.

Each week, participants reported some issues and feature requests. We regularly improved the tool accordingly and released five new versions (`0.2.0` to `0.2.5`) in total during the study. Regularly improving LightShare helped us keep participants involved in the experiment. Table 6.3 briefly describes those versions (for more details check the LightShare issue tracker [87]).

There were bug reports affecting some users using LightShare on the first day. We addressed them on the same day (version `0.2.1`). C02 user had Two-Factor authentication enabled on his Discord account that was not supported by LightShare during the first week of the study. C02 and C10 started using the tool the second week as we added this support (version `0.2.2`). Participants also asked to extend LightShare support to other text and code editors which we introduced the 3rd week (version `0.2.3`). Those changes did not change the LightShare workflow described in Section 6.2.2, except of the fact that starting the third week, they were able to initiate sharing from any text and source code editor, not only from the Playground. We did not observe those new use cases on the public Pharo channels and we were not able to read private communications.

Release	Date	Description
0.2.0	March 12 (1st week)	Initial study version
0.2.1	March 12 (1st week)	Bugfixes reported by participants
0.2.2	March 19 (2nd week)	Add Two-Factor authentication support (affecting C02 and C10)
0.2.3	April 3 (3rd week)	Share any text or code editor
0.2.4	April 4 (4th week)	Minor fixes not affecting participants
0.2.5	April 9 (4th week)	Minor fixes not affecting participants

Table 6.3: LightShare releases during the study period between March 12 and April 15, 2018.

6.3.3. Analysis

We used several data sources for the analysis of the results. Initial, weekly, and final Surveys are available at [97].

Initial survey. We asked participants to fill out the initial survey the first day of study. About the half of the participants filled it out during the first two days, the rest mostly a week later. We asked for the demographic information and for electronic communication habits.

Weekly survey. This survey was shared at the beginning of each week. We asked participants how many days during the last week they programmed, communicated on Discord, and used LightShare. This was followed by questions about the usefulness of LightShare, a situation where LightShare was useful, a situation where it was not possible to use it, and new recommendations to improve the tool and communication experience.

Dedicated channel and personal conversations. We used a dedicated Discord community channel to promote, discuss, and help using LightShare. Participants used it for testing the application, sharing their thoughts, issues, and recommendations. The channel was particularly useful for maintaining interest about the tool among participants.

Usage log. The usage of LightShare was logged, the data was anonymous. We recorded anonymous user ID, a flag describing whether a message was sent to a Discord channel or directly to as user, the type of artifact that was shared, and the message creation date and time. We were able to track what artifacts users shared on public Discord channels only, and we report on some of these usages in Section 6.4.

Final survey. We sent a final survey to the registered participants, in which we asked them to confirm several facts that we uncovered during the analysis. This survey was also an additional opportunity to gather qualitative feedback.

Public discussion. We manually analyzed Pharo public channels to find particularly interesting patterns that show how LightShare was used. We were able to analyze all public discussions that involved LightShare usage. We did not have access to LightShare messages that were sent to private and company channels.

6.3.4. Limitations of the Study

This study is an initial evaluation of the communication tool LightShare. To increase confidence in our findings, we used three data sources: tool usage logs, publicly available messages, and weekly surveys. In addition, we conducted a confirmatory survey at the end of the study and double checked our understanding in personal chats with each participant. Nevertheless, this study has some limitations.

The thesis author performed the data collection, contacted and kept discussion with participants. None of the participants is a close friend of the thesis author (or his supervisors). However, the authors contribute to the Pharo community and therefore are known to participants, a general respect or authoritative position of the thesis author might affect how participants responded to our surveys. We did not observe such behavior as we received both positive and negative ratings, some questioning the meaningfulness of integrated chats in general, but we cannot guarantee its absence. Seven participants accepted to be enrolled in the study, but nevertheless they did not perform any activity (see participants without the $*$ character in Table 6.2). One reason might be that they had no time, another reason that they did not find LightShare interesting, but were unwilling to say it.

Our case study was carried out in a community around the Pharo development environment. This may limit the results to this particular community and implies that our findings may not apply to commercial environments or in other software communities. Further studies in different environments will complement this study and improve the LightShare usability understanding.

Our study was conducted over a five-week time period. Our participants were engaged with their business and private affairs and therefore their main priority was not using LightShare. As a consequence this could have the effect of not revealing some of the strengths and weaknesses of the tool, and the study may be incomplete. To mitigate this effect and to keep participants engaged in the study, we kept regular personal discussion with them, sent them weekly questionaries (that also served as reminders), and released five LightShare updates during the study, reflecting their feature proposals and bug-reports. In addition, we offered them participation in a raffle about a Smalltalk book of their choice.

Conducting the study in a real environment where participants have their work responsibilities does not allow much space to explore completely new developer behavior habits. This study should be therefore interpreted as an incremental improvement of existing communication issues. A different study setup should likely be used to explore completely new ways of communication that disrupt the way developers coordinate.

It may be the case that participants of the LightShare study used Pharo tools differently compared to what we observed in Chapters 3 and 4. It may also be case that they neither share similar communication habits compared to our pre-study survey. This would imply that they could have difficulties effectively applying LightShare during their daily work. However, we have observed interesting LightShare use cases and think that it was little or no problem. In addition, we handed over LightShare documentation at the beginning of the study, which simplified the LightShare integration into the daily practice.

6.4. Results and Findings

We first present general usage statistics of LightShare, the types of shared artifacts, and compare results with Discord usage. We then present the different types of scenarios in which LightShare was used with examples.

6.4.1. Participant Activity

Overall activity. Figure 6.6 compares participant activity during two periods, including 15 participants who actually used LightShare. The gray bar shows the number of messages sent by individual participants during the 3-month period (from November 2017 to January 2018). This is the period we used to contact most active users. Displayed (gray bar) values represent average values corresponding to 5 weeks of activity to be able to compare it with the 5-week study period (black bar values).

We compare it with this period to see if the participant activity was similar during both time periods.

We observe that the participant activity varied contrasting those time periods. Some participants were more active during the 3-month period, some of them during the study period. On average, participants sent 25% more messages during the study period.

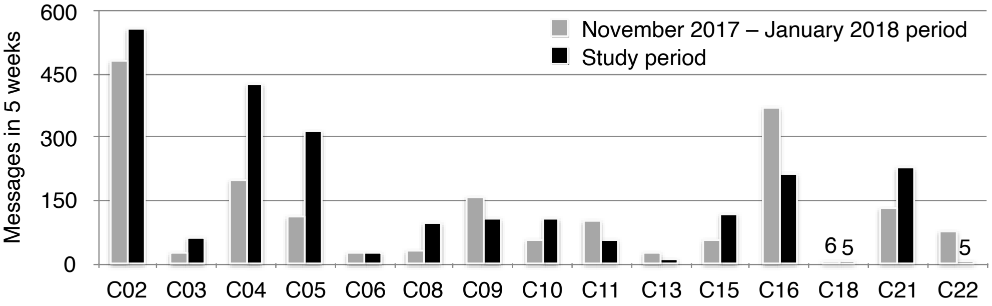


Figure 6.6: Average number of messages sent by 15 participants during the 3-month period and the study period in Pharo Discord channels.

The following results in this section combine observations of the 15 participants we contacted who actually used LightShare, and 6 other users who discovered and used the tool on their own initiative, for a total of 21 users in the five week period between March 12, 2018 and April 15, 2018.

Weekly activity. Figure 6.7 compares participant activity during the study period by sending messages using a standard Discord client (gray bars) to Pharo public channels and by posting messages using LightShare (black bars). The y-axis uses logarithmic scale to make all data readable. We observe that participants gradually reduced their activity in both cases, using the standard Discord client and using LightShare.

Figure 6.8 shows the ratio between messages sent each week using LightShare and the standard client. We observe that the less active users were, they used LightShare less frequently. One reason could be that participants had less opportunities to use LightShare

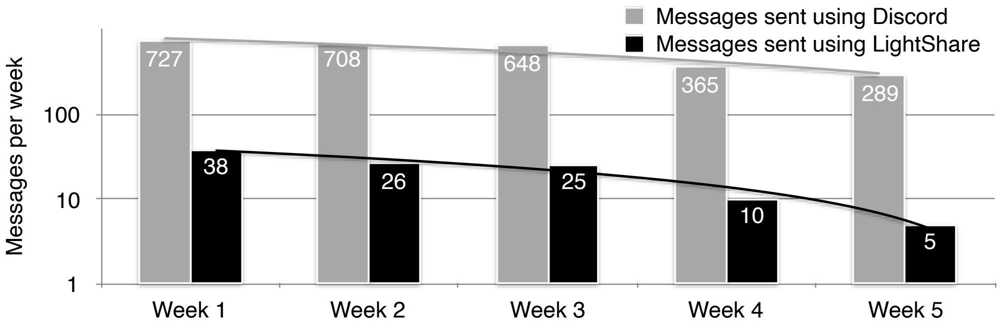


Figure 6.7: Messages sent each week by 15 participants and 6 other users during the study period.

as they were less active. Another reason could be that participants were losing interest in LightShare. The risk of losing interest was a challenging part of this study as participants worked on their daily projects and duties and the study was a voluntary commitment that might dilute while facing their real project challenges.

The presented trends suggest that there was less participant activity by the end of the study. This also explains why we noticed a downward LightShare use, as LightShare helps to initiate particular conversations, which do not arise every time users discuss a subject. On the other hand, we cannot ignore the fact that participants simply did not need to continue using the tool. In both cases, this reduces the chances of clarifying the tool advantages and disadvantages.

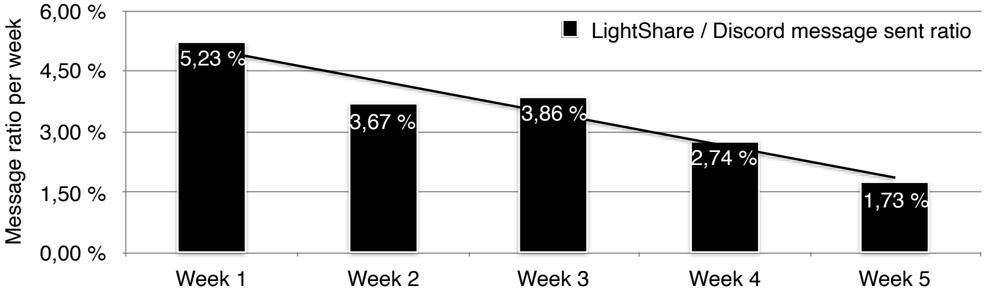


Figure 6.8: Ratio of messages sent each week by 15 participants and 6 other users during the study period using LightShare / Discord.

Sending messages. Participants used LightShare 104 times, from which in 36 cases, they sent messages to public Pharo channels, 6 to individuals, and 62 to other channels (Figure 6.9). The other channels are Discord servers that were not available to us, *e.g.,* company private discussions.

Figure 6.10 shows that nine participants used LightShare five or more times (at least once a week on average). Usage patterns vary, with some participants (e.g. C04, U28) posting exclusively on non-public channels, while other (e.g. C02, C16) published mostly in Pharo channels.

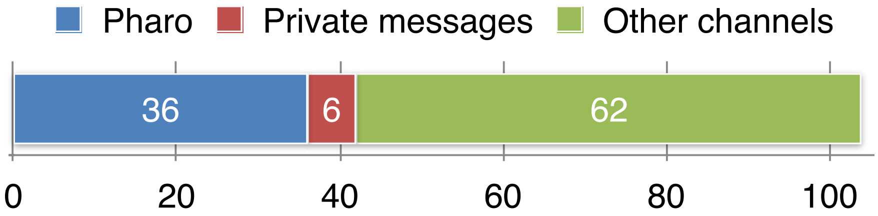


Figure 6.9: Messages sent by LightShare to public Pharo channels, to individuals, and to other channels.

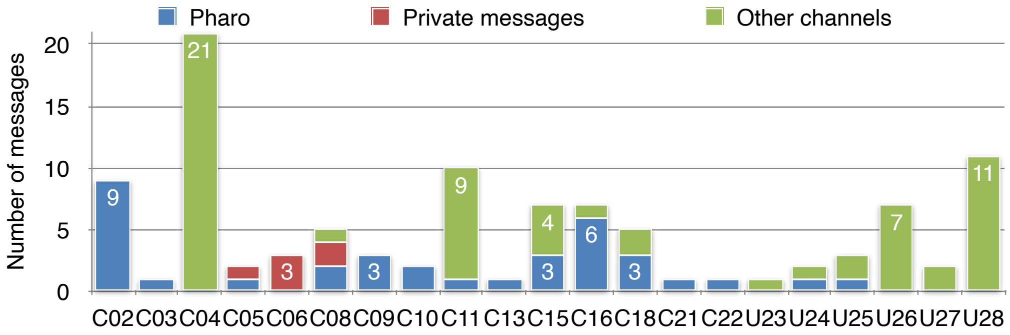


Figure 6.10: Participant messages sent by LightShare to public Pharo channels, to individuals, and to other channels.

Loading artifacts. Messages shared using LightShare have an option to load the shared artifacts to the IDE (see Section 6.2, Figure 6.3). We found five occurrences of this usage, thus this feature was used infrequently. Our log, which contains information about loading artifacts to the IDE, is anonymous, thus we do not know who did it and what he or she loaded.

6.4.2. Message Content

In this section we discuss what information was shared using LightShare. Messages can contain text, code snippets, and attachments, *e.g.,* screenshots, GIF images, source code packages. Figure 6.11 shows a breakdown of the content kinds that were shared with LightShare. Our first observation is that 89% (93 out of 104) of messages do contain an artifact; only 11% (11 out of 104) do not. In addition, close to half of the messages (51 out of 104) include more than one artifact (such as a code and a screenshot). This leads us to conclude that LightShare reached its goal of promoting the artifact sharing.

Code snippets. Figure 6.11 shows that 85 out of 104 (82%) messages include a code snippet. Participants sent 31 out of 85 messages to Pharo channels. Those messages included code snippets with up to 10 ten lines of code. We read (but not executed) the publicly shared code snippets and they appeared to be complete and thus executable. Some concrete scenarios are described in Section 6.4.4.

Attachments. Figure 6.12 shows that 59 out of 104 (57%) messages include attachments. In most cases, the attachments are screenshots (57 out of all 104 LightShare messages, which is 55%; and 57 out of all 59 attachments, which is 97%).

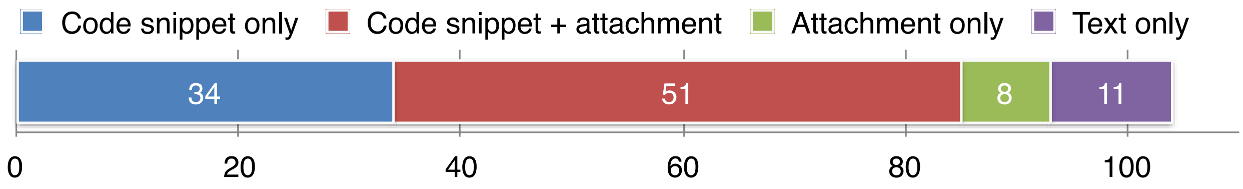


Figure 6.11: Shared artifacts: snippets and attachments.

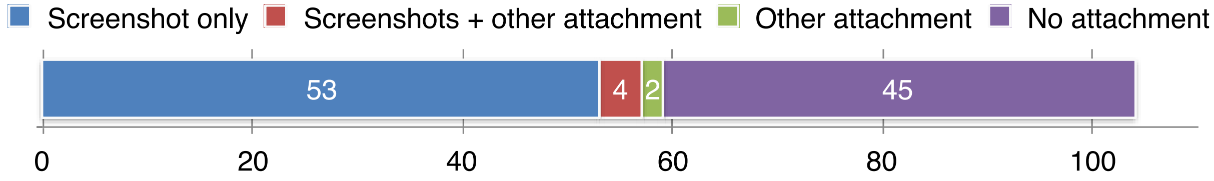


Figure 6.12: Shared artifacts: attachment types.

An important portion of the screenshots (21 out of 57) were attached to messages sent to Pharo channels, allowing us to better understand their type. Figure 6.13 shows that the prevalent use case is screenshots including shared playgrounds. The playground typically includes an elaborated code snippet and the code snippet result in an integrated object inspector [19].

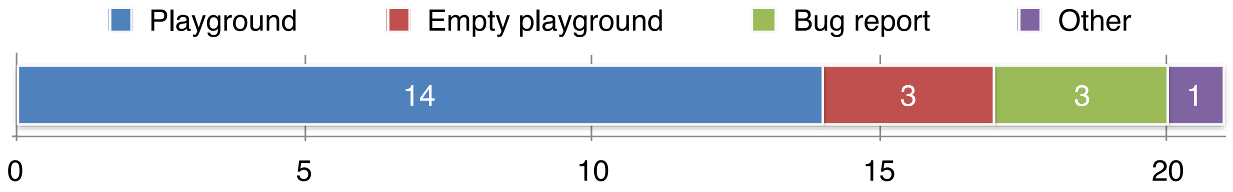
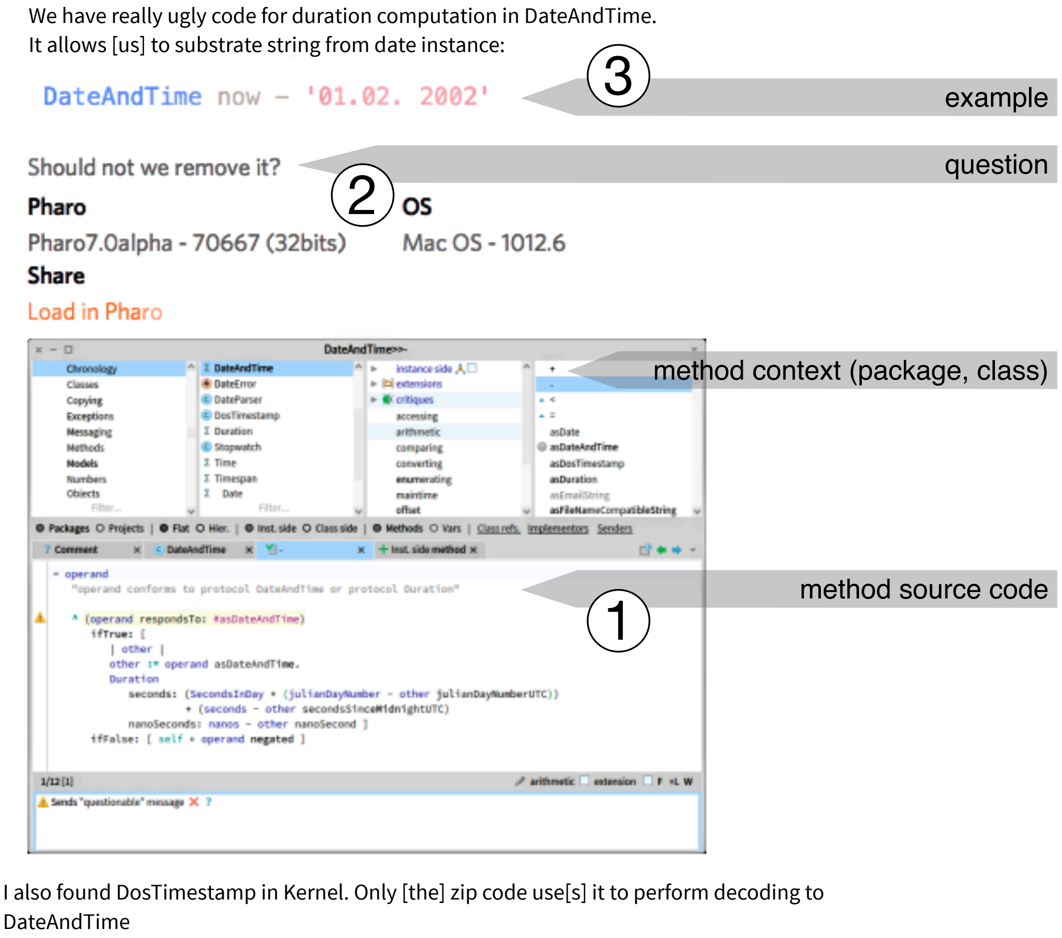


Figure 6.13: Shared screenshot attachments.

We found three empty screenshots (a playground with no content). Such screenshots do not offer any additional information to readers and may have a confusing effect in a conversation. LightShare should automatically detect such scenarios and exclude (or warn the user). In three cases, users reported a bug: two cases included a debugger screenshot, and the third a screenshot of a defective graphical widget (more details in Section 6.4.4). Finally, we observe one case where C05 shared a screenshot of a source code editor (see Figure 6.14): the developer wanted to show the source code of a method (part 1), proposing to remove it from the Pharo core system as he considered it *ugly*[1] (part 2). In addition, he shared a code snippet expressing how the method is used (part 3).

Other attachments. Participants C04, C06, and user U26 shared in four cases animated GIFs. Unfortunately, we do not have access to those messages to closely analyze their content, as they were not shared on public Pharo channels. C11 and C18 sent once a message with enclosed source code. We asked them for details about it. C11 reported that he tested the feature. C18 did not remember he attached the source code.

[1] We think that C05 used the expression *ugly code* to express that the particular method is problematic and does *not* follow the recommended programming techniques, *e.g.,* Software Design Patterns [57]



We have really ugly code for duration computation in DateAndTime.
It allows [us] to substrate string from date instance:

```
DateAndTime now - '01.02. 2002'
```

Should not we remove it?

Pharo Pharo7.0alpha - 70667 (32bits)

OS Mac OS - 1012.6

Share

Load in Pharo

```
- operand
  "operand conforms to protocol DateAndTime or protocol Duration"

  ^ (operand respondsTo: #asDateAndTime)
    ifTrue: [
      | other |
      other := operand asDateAndTime.
      Duration
        seconds: (SecondsInDay * (julianDayNumber - other julianDayNumberUTC))
                 + (seconds - other secondsSinceMidnightUTC)
        nanoSeconds: nanos - other nanoSecond ]
    ifFalse: [ self + operand negated ]
```

Sends "questionable" message

I also found DosTimestamp in Kernel. Only [the] zip code use[s] it to perform decoding to DateAndTime

Figure 6.14: Participant proposed to delete a method that was considered *ugly*.

Users also reported difficulties recording animated GIFs [23], which may explain its low usage. We were not able to target this issue during the study period. On other hand, users also proposed that sharing animated GIFs would be useful for Twitter micro-blogging [24], which suggest that deploying a solid way for recording and sharing animated GIFs may change the way users could use LightShare and animated GIFs.

Summary. The message content analysis shows that participants predominately shared playground code snippets and the associated result screenshots. Some participants tested sharing animated GIFs and found usability issues that likely prevented its wider use during the study. As almost half of the LightShare messages include more than one artifact, typically a code snippet and a screenshot, we conclude that the LightShare prototype reached its goal of promoting the artifact sharing.

6.4.3. Comparison with Discord Sharing

While the usage of LightShare was infrequent, it is useful for comparing it to sharing activity in general. During the 5-week period prior to the study, participants posted 1,883 messages to Discord. They posted code snippets in 86 messages (5%) and attached a screenshot in 46 messages (2%). During the 5-week study period, our participants sent 2,737 messages on public Pharo channels. They shared code snippets without using LightShare in 86 messages (3%) and screenshots in 48 messages (2%). Note that the amount of sharing without using LightShare is comparable between the two periods. During the study period, additional 31 code snippets and 18 non-empty screenshots were shared on public Pharo channels with LightShare. Thus, 27% of code snippets and non-empty screenshots were shared with LightShare by our participants on public channels.

Participant C02 was the only one who shared a different attachment than a screenshot without using LightShare during the study period. C02 shared a 16-second long video (without audio) presenting a new feature he implemented. The most similar LightShare usages are the four messages including animated GIFs.

6.4.4. Usage Scenarios

We looked at the Pharo public channel conversations where LightShare was used. We identify four scenario types in which our participants used LightShare:

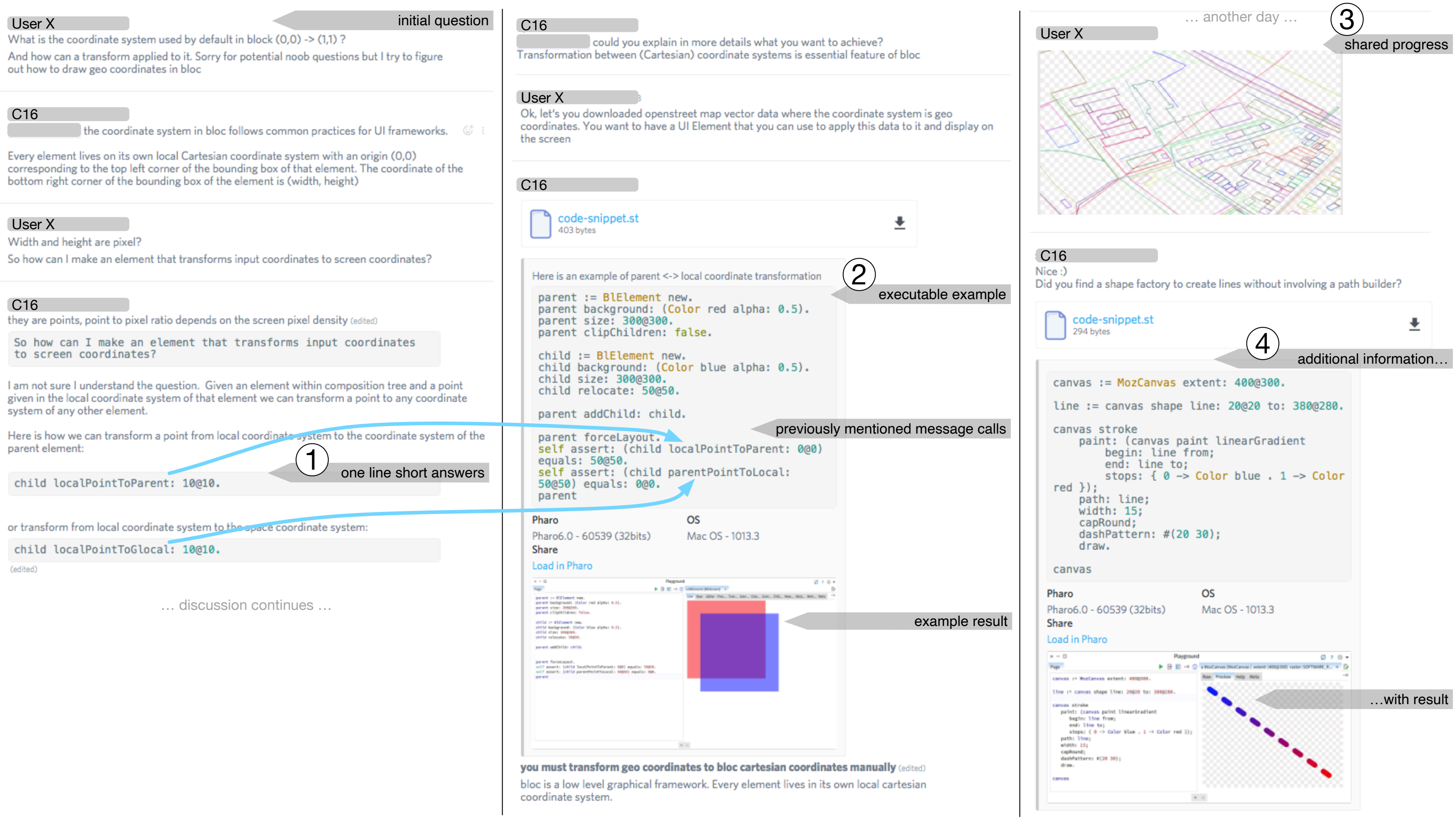
- *Early feedback* by sharing explicit information and hence increasing message readability, instead of incomplete messages where readers must hypothesize an overall story, *e.g.,* missing code snippet parts and code snippet results.
- *Learning tools* by sharing tool screenshots that accompanied discussions, explicitly pointing where (or how) information can be found.
- *Helping beginners* by sharing complete code snippets and the code snippet execution results, instead of implicitly suggesting which libraries to use for the particular tasks.
- *Reporting bugs* by providing explicit information, *e.g.,* operating system and Pharo versions, stack traces, screenshots, and error-reproducing code snippets.

In the identified use cases, users transformed the text-oriented discussions to the level of Live Programming, making messages easy to understand and moreover easy to reproduce on their local programming environments (all this without having a fully integrated chat client). In other words, more readable messages provided an early feedback while reading posts as those posts include explicit (complete) information, giving readers change to follow discussions. In the opposite situation, they would have to deal with implicit information and reconstruct mental models and code snippets to be able to follow discussions. In addition, users utilized their favorite Live Programming tools (as revealed in Chapter 4) to share information, *e.g.,* playgrounds and object inspectors.

Early feedback. Figure 6.1 illustrates how the additional information provided by LightShare increases the message readability: readers can see both a code snippet (with syntax highlighting), and its result, without executing it. Readers therefore have a clearer and more accurate picture of the discussed situation. As Live Programming users benefit from the *early feedback* (see Chapter 2), LightShare message readers benefit from the provided explicit information. LightShare messages reach the level one of liveness (being informative, see Chapter 2) as readers cannot manipulate the posted information, however the information is accurate and explicit. In addition, users wishing to reach the level four of liveness can use LightShare to load and execute code snippets in their IDEs. Indeed—and as foreseen—a large part of the messages that we analyzed correspond to this early feedback use case.

For instance, during the study C16 used LightShare seven times, of which four were similar to the motivation scenario. C16 is the author of a graphics library. These examples happened while he was explaining how to transform coordinates between two coordinate systems, see Figure 6.15:

- Early in the conversation, he attempted to respond by sharing short (one line) code snippets (part 1), and by mentioning API element names and examples in the conversation, without using LightShare. He described how to do the task in abstract terms.
- As it was still unclear to the user, C16 shared an example of the differences between parent and child coordinates, with screen captures of the results, with LightShare (part 2). The aforementioned methods (one-liners, part 1) were thus supplemented with all the necessary code for their full use (readers thus could run the code).
- He also added an additional example where a more complex transformation was specified, with a screen capture of the result, again with LightShare (omitted from the figure to simplify the use case).
- The user was satisfied, and shared the next day the result of his work (a rendering of a map, without using LightShare, part 3).
- C16 then (unprompted) provided an example of an alternative API to draw lines with an accompanying screen capture, again with LightShare (part 4).
- Finally, he followed up by sharing a final example of how to export the map rendering to an SVG file format, still with LightShare (omitted from the figure to simplify the use case).



User X
What is the coordinate system used by default in block (0,0) -> (1,1) ?
And how can a transform applied to it. Sorry for potential noob questions but I try to figure out how to draw geo coordinates in bloc

C16
[redacted] the coordinate system in bloc follows common practices for UI frameworks.

Every element lives on its own local Cartesian coordinate system with an origin (0,0) corresponding to the top left corner of the bounding box of that element. The coordinate of the bottom right corner of the bounding box of the element is (width, height)

User X
Width and height are pixel?
So how can I make an element that transforms input coordinates to screen coordinates?

C16
they are points, point to pixel ratio depends on the screen pixel density (edited)

```
So how can I make an element that transforms input coordinates
to screen coordinates?
```

I am not sure I understand the question. Given an element within composition tree and a point given in the local coordinate system of that element we can transform a point to any coordinate system of any other element.

Here is how we can transform a point from local coordinate system to the coordinate system of the parent element:

```
child localPointToParent: 10@10.
```

or transform from local coordinate system to the space coordinate system:

```
child localPointToGlocal: 10@10.
```

(edited)

… discussion continues …

C16
[redacted] could you explain in more details what you want to achieve?
Transformation between (Cartesian) coordinate systems is essential feature of bloc

User X
Ok, let's you downloaded openstreet map vector data where the coordinate system is geo coordinates. You want to have a UI Element that you can use to apply this data to it and display on the screen

C16
code-snippet.st
403 bytes

Here is an example of parent <-> local coordinate transformation

```
parent := BlElement new.
parent background: (Color red alpha: 0.5).
parent size: 300@300.
parent clipChildren: false.

child := BlElement new.
child background: (Color blue alpha: 0.5).
child size: 300@300.
child relocate: 50@50.

parent addChild: child.

parent forceLayout.
self assert: (child localPointToParent: 0@0)
equals: 50@50.
self assert: (child parentPointToLocal:
50@50) equals: 0@0.
parent
```

Pharo	OS
Pharo6.0 - 60539 (32bits)	Mac OS - 1013.3

Share
Load in Pharo

you must transform geo coordinates to bloc cartesian coordinates manually (edited)
bloc is a low level graphical framework. Every element lives in its own local cartesian coordinate system.

… another day …

User X

C16
Nice :)
Did you find a shape factory to create lines without involving a path builder?

code-snippet.st
294 bytes

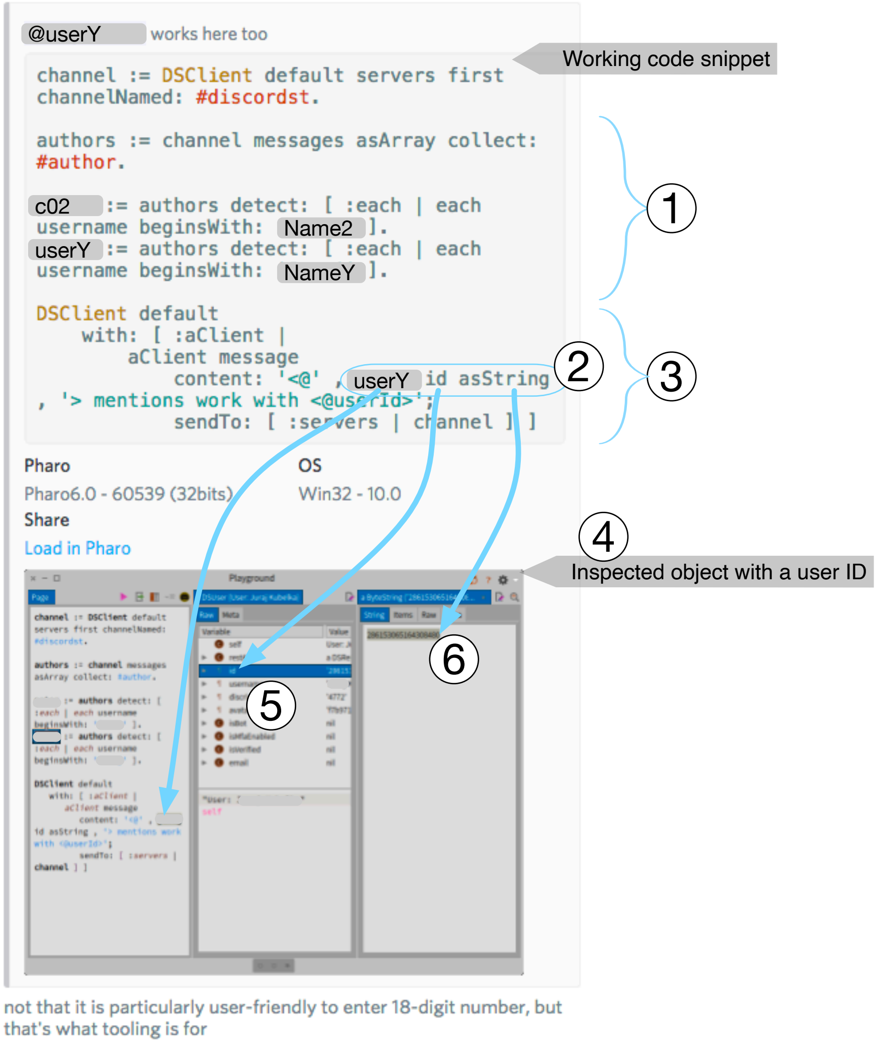
```
canvas := MozCanvas extent: 400@300.

line := canvas shape line: 20@20 to: 380@280.

canvas stroke
    paint: (canvas paint linearGradient
        begin: line from;
        end: line to;
        stops: { 0 -> Color blue . 1 -> Color
red });
    path: line;
    width: 15;
    capRound;
    dashPattern: #(20 30);
    draw.

canvas
```

Pharo	OS
Pharo6.0 - 60539 (32bits)	Mac OS - 1013.3

Share
Load in Pharo

Figure 6.15: C16's discussion on a graphical library coordinate system.

Participant C02 is another example. C02 was exploring how to programmatically post Discord messages with user mentions (see Figure 6.16):



@userY works here too

```
channel := DSClient default servers first
channelNamed: #discordst.

authors := channel messages asArray collect:
#author.

c02 := authors detect: [ :each | each
username beginsWith: Name2 ].
userY := authors detect: [ :each | each
username beginsWith: NameY ].

DSClient default
    with: [ :aClient |
        aClient message
            content: '<@' , userY id asString
, '> mentions work with <@userId>';
            sendTo: [ :servers | channel ] ]
```

Pharo
Pharo6.0 - 60539 (32bits)

OS
Win32 - 10.0

Share
Load in Pharo

not that it is particularly user-friendly to enter 18-digit number, but that's what tooling is for

Figure 6.16: C02's message explaining how to programmatically post messages including user mentions.

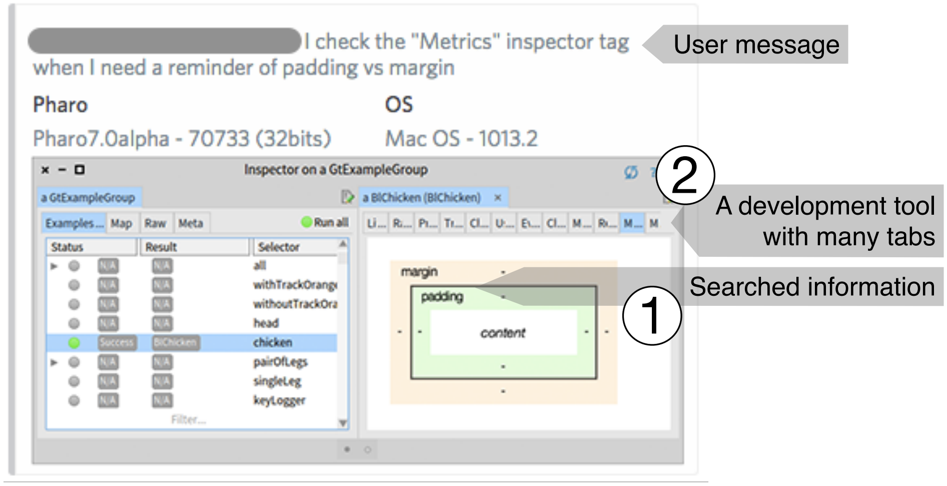
- He identified a limitation of LightShare and wanted to provide a solution for it.
- He shared his findings by posting a code snippet that queries Discord members (part 1), searches a member ID (part 2), and sends it to Discord (part 3).
- Since the member ID was the important information, he executed the code snippet and emphasized its location in the resulting object graph (part 4).
- He then navigated in the object graph until he found the member ID field (part 5), and highlighted its value (part 6).

He then shared both the code snippet, and a screenshot of the object graph with the highlighted member ID. Thus, this screenshot illustrates better where the member ID is. The same information is less readable in the code snippet (part 2) as it is wrapped by other lines of code (parts 1 and 3).

Learning tools. Participant C05 wanted to recall what margin and padding means in a graphical environment, see Figure 6.17.

- C05 said "*I forget what the difference [is] between margin and padding?*" and received the answer "*margin = outside, padding = inside*".
- C05 thanked to the user who sent the answer and continued "*I know it is well known but [. . .] I always [forget] the difference [between margin and padding].*"
- C15 answered, using LightShare, "*I check the "Metrics" inspector [tab] when I need a reminder of padding vs. margin.*" (part 1)

The message includes a development tool window to explicitly show where to find the Metrics tab (part 1). This is very helpful, since the inspector has many tabs (part 2), causing discoverability issues.



[redacted] I check the "Metrics" inspector tag when I need a reminder of padding vs margin

Pharo: Pharo7.0alpha - 70733 (32bits)

OS: Mac OS - 1013.2

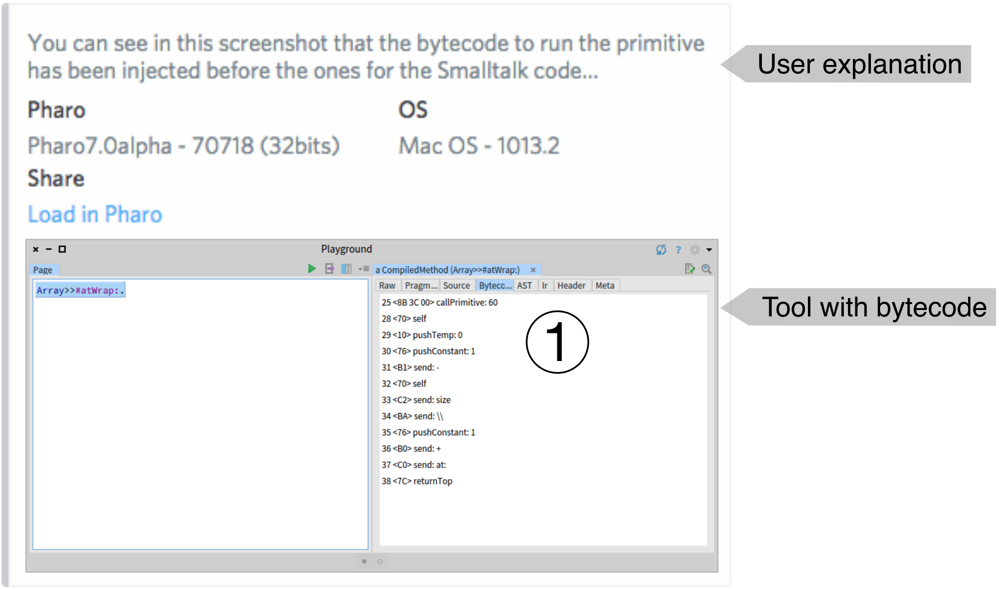
Inspector on a GtExampleGroup

Status	Result	Selector
N/A	N/A	all
N/A	N/A	withTrackOrange
N/A	N/A	withoutTrackOra
N/A	N/A	head
Success	BlChicken	chicken
N/A	N/A	pairOfLegs
N/A	N/A	singleLeg
N/A	N/A	keyLogger

Figure 6.17: C15's message explaining where in the IDE to find margin and padding information.

Similarly C15, in another conversation, explained how pragmas (Pharo source code annotations) are processed by the Pharo compiler and inserted into the method bytecode: "*. . . pragma [, an annotation,] of the form <primitive:. . . > is processed by the [Pharo] compiler. . . [to] the bytecode to run the primitive [. . . and] injected before [. . . a] Smalltalk code [of a discussed method]*". He then shared a Pharo tool screenshot showing the bytecode, see Figure 6.18 part 1. Readers can thus see what the bytecode looks like, and in addition, can find out how to display the bytecode of any other method.

Campbell and Miller observed that tool awareness is a problem in IDEs that are used by software developers [15]. Murphy-Hill and Murphy conducted series of interviews in industry to explore how software development tool discovery happens from peers. They found that the tool discovery happens infrequently, yet programmers can effectively learn from their co-workers both in co-located and remote settings [119]. Google uses a technique called "Testing on the Toilet," to improve tool awareness [8]. It consists of one-page printed newsletters, written by developers and posted in restrooms. Emerson Murphy-Hill *et al.* evaluated this technique and they suggest that the technique is generally effective at increasing tool awareness [121].



You can see in this screenshot that the bytecode to run the primitive has been injected before the ones for the Smalltalk code...

Pharo: Pharo7.0alpha - 70718 (32bits)

OS: Mac OS - 1013.2

Share: Load in Pharo

```
Array>>#atWrap:.
```

```
25 <8B 3C 00> callPrimitive: 60
28 <70> self
29 <10> pushTemp: 0
30 <76> pushConstant: 1
31 <B1> send: -
32 <70> self
33 <C2> send: size
34 <BA> send: \\
35 <76> pushConstant: 1
36 <B0> send: +
37 <C0> send: at:
38 <7C> returnTop
```

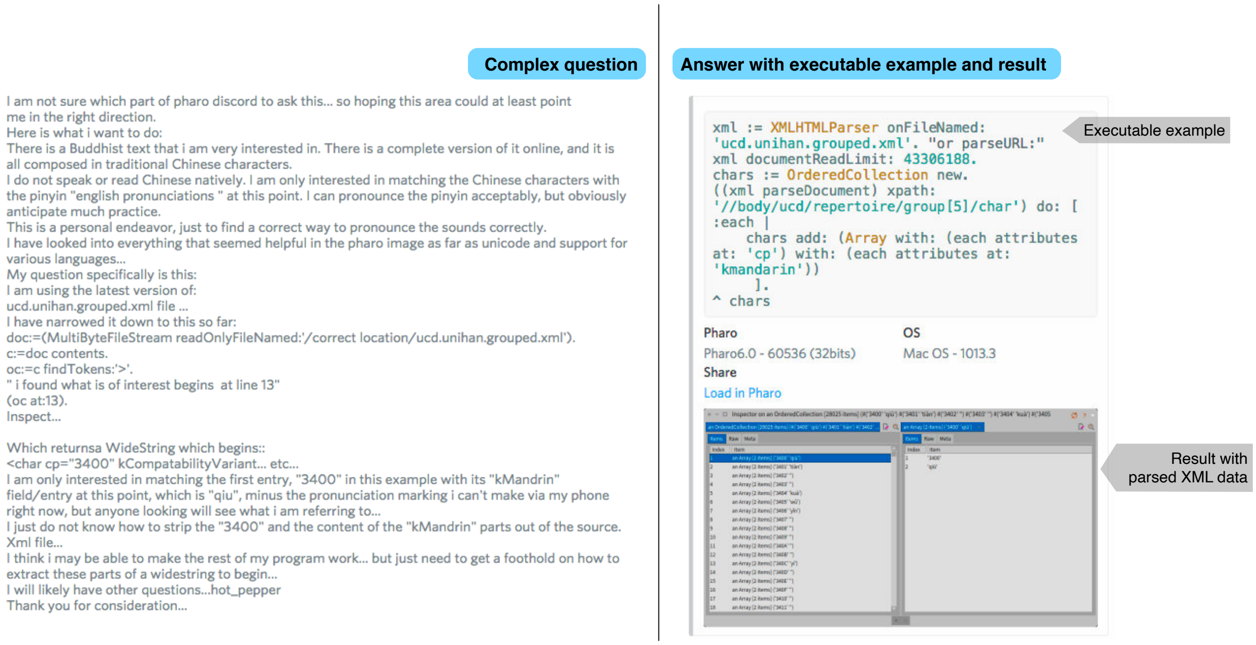
Figure 6.18: C16's code snippet result explaining compiled code details.

Helping beginners. A beginner posted a complex message trying to explain how he struggles to extract information from an XML file, see the left part of Figure 6.19:

- The message had 32 lines of mixed text and code parts. This could be a reason why the question was not answered and disappeared under the flood of new messages.
- The user asked it again stating "*I am still trying to merely extract certain parts from a huge .xml file... [I] posted above. I still do not find how to do it[...]*"
- He obtained several recommendations to use an XML parser and general hints how to deal with the task. Those messages did not include any code snippet example.
- The beginner still struggled with the task, closing his thoughts with the statement "*Is this really [...that] difficult?*"
- Then U25 joined the discussion and posted a working code snippet with a screenshot displaying results, see the right part of Figure 6.19.
- The beginner later answered "*You are a great example of why this community is so incredible: you helped me exactly where and when it was needed... The code snippet worked perfectly.*"

Being a beginner, executable and correctly written code was crucial to him, as he did not know how to use a XML parser. User U25, in addition, shared a screenshot of his programming environment that showed how to inspect the parsed XML model, and how to navigate inside the XML model to the desired information (similarly to C15 in the previous example).

Reporting bugs. Some participants used LightShare to report bugs, see Figure 6.20. C09 sent a window screenshot pointing to incorrectly displayed buttons, saying "*There seem to be some buttons cut-off.*" (part 1). In another instance, C09 reported a bug that included a code snippet producing an error (part 2) and a screenshot of the resulting debugger window with a stack trace of the error (part 3).



I am not sure which part of pharo discord to ask this... so hoping this area could at least point
me in the right direction.
Here is what i want to do:
There is a Buddhist text that i am very interested in. There is a complete version of it online, and it is all composed in traditional Chinese characters.
I do not speak or read Chinese natively. I am only interested in matching the Chinese characters with the pinyin "english pronunciations " at this point. I can pronounce the pinyin acceptably, but obviously anticipate much practice.
This is a personal endeavor, just to find a correct way to pronounce the sounds correctly.
I have looked into everything that seemed helpful in the pharo image as far as unicode and support for various languages...
My question specifically is this:
I am using the latest version of:
ucd.unihan.grouped.xml file ...
I have narrowed it down to this so far:
doc:=(MultiByteFileStream readOnlyFileNamed:'/correct location/ucd.unihan.grouped.xml').
c:=doc contents.
oc:=c findTokens:'>'.
" i found what is of interest begins at line 13"
(oc at:13).
Inspect...

Which returnsa WideString which begins::
<char cp="3400" kCompatabilityVariant... etc...
I am only interested in matching the first entry, "3400" in this example with its "kMandrin" field/entry at this point, which is "qiu", minus the pronunciation marking i can't make via my phone right now, but anyone looking will see what i am referring to...
I just do not know how to strip the "3400" and the content of the "kMandrin" parts out of the source. Xml file...
I think i may be able to make the rest of my program work... but just need to get a foothold on how to extract these parts of a widestring to begin...
I will likely have other questions...hot_pepper
Thank you for consideration...

```
xml := XMLHTMLParser onFileNamed:
'ucd.unihan.grouped.xml'. "or parseURL:"
xml documentReadLimit: 43306188.
chars := OrderedCollection new.
((xml parseDocument) xpath:
'//body/ucd/repertoire/group[5]/char') do: [
:each |
    chars add: (Array with: (each attributes
at: 'cp') with: (each attributes at:
'kmandarin'))
     ].
^ chars
```

Pharo
Pharo6.0 - 60536 (32bits)

OS
Mac OS - 1013.3

Share

Figure 6.19: U25's code snippet result.

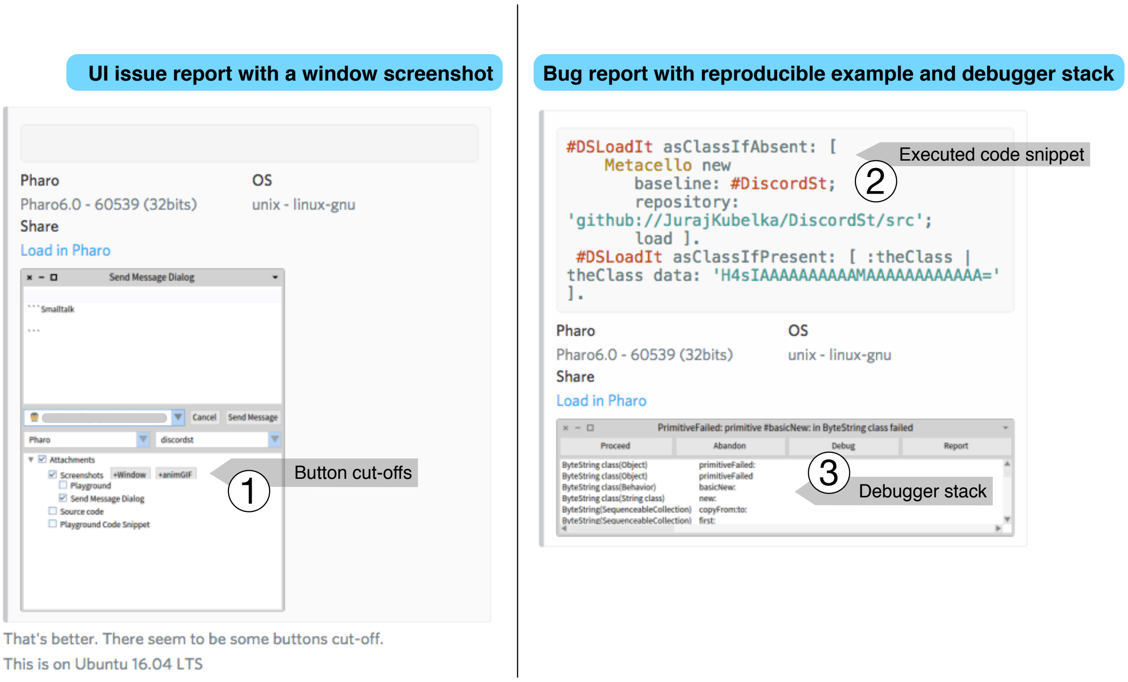
To make LightShare messages more complete, useful information about the operating system and the Pharo versions is also included. This feature is inspired by a report by Zimmermann *et al.* who observed that using communication tools separated from development environments makes it difficult to share important information in bug reports (as it is difficult and time consuming) [169].

Summary. We identify four scenario types in which LightShare is particularly useful. Users can more easily share explicit information and therefore improve message readability. Readers can learn tools as developers share their tool screenshots while explaining where and how information can be reached. Beginners can benefit from complete answers with executable code snippet examples. Bug reporters can share additional (and fundamental) information while reporting on application issues.

More readable messages give readers change to follow discussions as they include explicit (complete) information. LightShare messages therefore provide an early feedback while reading them, just like Live Programming users benefit from early feedback while exploring running applications.

6.5. Feedback and Discussion

We start by discussing the feedback we obtained from participants, before discussing possible ways to follow up our work. The feedback we obtained comes from weekly surveys (filled on average by 10 participants), a final survey (filled by 11 participants), and informal discussion with participants on the dedicated Discord channel or in private Discord discussions.



Pharo | OS
Pharo6.0 - 60539 (32bits) | unix - linux-gnu
Share
Load in Pharo

~~~
```Smalltalk

```
~~~

Pharo
discordst

☑ Attachments
☑ Screenshots
□ Playground
☑ Send Message Dialog
□ Source code
□ Playground Code Snippet

That's better. There seem to be some buttons cut-off.
This is on Ubuntu 16.04 LTS

```
#DSLoadIt asClassIfAbsent: [
    Metacello new
        baseline: #DiscordSt;
        repository:
'github://JurajKubelka/DiscordSt/src';
        load ].
 #DSLoadIt asClassIfPresent: [ :theClass |
theClass data: 'H4sIAAAAAAAAAMAAAAAAAAAAAA='
].
```

Pharo | OS
Pharo6.0 - 60539 (32bits) | unix - linux-gnu
Share
Load in Pharo

PrimitiveFailed: primitive #basicNew: in ByteString class failed

```
ByteString class(Object)                   primitiveFailed:
ByteString class(Object)                   primitiveFailed
ByteString class(Behavior)                 basicNew:
ByteString class(String class)             new:
ByteString(SequenceableCollection)         copyFrom:to:
ByteString(SequenceableCollection)         first:
```

Figure 6.20: C09's bug reports

6.5.1. On Sharing Artifacts

Information clarity. There was a broad agreement that the information that LightShare provided was complete and clear. In the weekly surveys, we asked whether the participants saw a LightShare shared artifact during the previous week, and if so, whether it was complete and clear. We asked about the information clarity four times in total. Table 6.4 shows individual answers for each week and summaries for each participant and week. Individual symbols have the following meaning: 😄 is very complete and clear, 🙂 is somewhat complete and clear, ☹ is somewhat incomplete, and 😵 means that a participant did not read a messages sent by LightShare. No participant used the "very incomplete" option. Empty cells in the table stand for incomplete surveys. The summary column includes the most negative answer that a participant completed during the study.

Nineteen participants responded at least once to the weekly survey, with 5 responding that they did not see any artifact. Of the remaining 14 participants, 13 responded that the information was "somewhat" or "very" complete in all the surveys that they took. The only dissenting opinion (responding once "somewhat complete", and once "somewhat incomplete") did not provide additional feedback as to what they felt was missing (although there was a free-text field for the feedback).

Screenshots. All ten final survey respondents (C02, C03, C05, C08, C09, C10, C11, C18, C21, C22) agreed that sharing screenshots using LightShare is useful. Some participants proposed improvements such as: being able to edit screenshots, blurring parts of them, and also being able to take screenshots of a window portion and outside of the development environment (the current version only allows us to take screenshots of windows that belong to the IDE). C02 proposes to make the window selection more intuitive by clicking on windows: "*[...] having to choose from a textual list of windows is rather slow and confusing as I almost*

Participant	Week 1	Week 2	Week 3	Week 4	Summary
C02	🙀	🙀	🙀	🙀	🙀
C03	😄				😄
C04	😄	😄			😄
C05	😄	😄	😄	🙀	😄
C06	🙂				🙂
C07	😄	😄	🙀		😄
C08	🙀	🙂			🙂
C09	🙂	😕	🙀		😕
C10	🙂	🙀		🙀	🙂
C11	🙂			🙀	🙂
C12	😄				😄
C13	🙀			🙀	🙀
C14				🙀	🙀
C15		🙂	🙂		🙂
C18	🙂				🙂
C19			🙀		🙀
C20	🙀				🙀
C21				😄	😄
C22	🙂	🙀		🙀	🙂
Total					
answers	15	9	6	8	19
🙂 or 😄	11	5	2	1	13
😕	0	1	0	0	1
🙀	4	3	4	7	5

Table 6.4: Participant weekly responses on information clarity: very complete and clear 😄, somewhat complete and clear 🙂, somewhat incomplete 😕, not read 🙀.

never look at window titles (this is not Pharo specific, I don't read them in OS [n]either, as it is almost always easier to identify [windows] by their content)."

Animated GIFs. Nine out of ten final survey respondents (C02, C05, C08, C09, C10, C11, C18, C21, C22) agreed that sharing GIFs is useful. C03 was undecided. C02, C05, C18 agreed that the feature needed improvements, such as a GIF editor, the ability to highlight zones of interest, and showing keyboard and mouse events. Several participants thought that animated GIFs can be used in other contexts, e.g., blog posts, and README documentation on GitHub. C02 expressed that "*[...] I needed a video/GIF for an external documentation, but didn't manage to quickly figure out how to use it [the LightShare animated GIF recorder] independently.[...]*"

C02 also expressed that "*I'm personally too inclined to show a static picture instead of a dynamic motion [animated GIF]. E.g.[,] if I need to show someone a sequence of steps they need to execute, I'd much rather use a screenshot/composition with arrows, so all the steps are visible at once.*" It suggest that developers have different preferences about shared information techniques and formats. On the other hand, C02 continues "*But I don't know* *what format is better for the reader.*"

Packages. Seven out of ten final survey respondents (C03, C05, C08, C09, C10, C18, C22) agreed that sharing packages is useful. C02, C11, C21 were undecided. C10 concludes that "*projects should share code via revision control [systems]. Sharing code [in chat] is very adhoc, so I'd expect its usefulness to be limited to 'hey can you check what's wrong with my code' [. . .]*". We agree with this sentiment; in fact, source code sharing with LightShare is aimed at this very scenario. A related issue is mentioned by C08: "*I would fear that loading a package [shared using LightShare] would disrupt my own [. . . source code].*"

This may explain why participants were reluctant to use the loading feature and needs to be improved in the future. One solution would be to provide a code preview. Also it is neither obvious what happens when users choose to load a LightShare message to their programming environments. The informing web page (see Figure 6.3) should be more explicit about it and the loading process should first report on message content before loading it.

New artifacts. Seven out of ten final survey respondents (C03, C08, C09, C10, C11, C21, C22) would like to be able to share additional artifacts. C02, C05, C18 are undecided. The most frequent mention is error stack traces (C03, C10, C11, C21). In a personal communication, C11 emphasized that providing accurate error information is difficult: "*when I get some error [. . .] and I have to ask about it on Discord, I cannot even copy the error message ([. . .] it is not a plain text [. . .]), so I have to rewrite it. Even if I do it, it is not enough. There is additional information that has to be shared (e.g. stack trace). [. . .] Almost every day someone asks questions on Discord about errors. So it would be good to have some button on the error window that would allow [posting of] a question on Discord with all information aggregated into it.*"

Similarly, Zimmermann *et al.* reported that developers are aware of the importance of describing "steps to reproduce a problem" in case of bug reports, but only a few provide them because they are difficult to provide [169]. LightShare actually has an experimental implementation of this feature, but it was not made available to participants; they resorted to sharing screenshots of the stack trace instead.

A variety of other kinds of artifacts were mentioned, with C03 saying: "*basically anything that has text/string should be shareable*". C03, C10 would want the flexibility of sharing arbitrary serialized objects (that could be then de-serialized in another IDE). C08 goes further and would like to be able to specify for each type of object the information that should be shared, and how: "*[. . .] I would like to be able to specify meta-information myself, that [LightShare] would automatically ship in its messages. [I want to include some] kind of generic (or unanticipated) type of content that I do not know [about] until I have the use case running.*" This is similar to how Pharo developers can define different object representations in tools such as the inspector [19], the debugger [21], and the search tool [152]. Users can extend those tools by providing definitions that determine how individual objects are presented, debugged, and searched.

Since the pre-survey responses presented in Section 6.2.3 indicate that sharing objects and debugger sessions (stack traces) is difficult, our original intention was to support this. Our future work is therefore to solve the technical challenges and provide a LightShare version that allows sharing these objects.

Summary. Overall, participants agree that LightShare is useful and LightShare messages are complete and clear (easy to follow). They consider screenshot sharing particularly useful. Sharing GIFs and packages (source code) is also considered useful (by less participants). For all the possible sharing features (artifacts) we received improvement recommendations, proposing also support for sharing objects and debugger stack traces.

6.5.2. On the Level of Integration

Conflicting opinions on integration. We find conflicting opinions on the level of integration that participants would like to have between Discord and Pharo. Early in the study, we found evidence that some users expected a full Discord client integrated in the IDE. C05 expected a full client, asking "*I cannot figure out how to open a window in Pharo with past messages from other users*". C19 also expected to be able to read messages within the IDE: he went as far as to implement a simple graphical interface prototype during the second day of the study for this task. C22 mentioned "*a full Discord client would be good*" in the second weekly survey.

On the other hand, one prospective participant declined to join the study, to avoid distraction "*[...] For that reason I enjoy having Discord as [a] separate client that I can silence when working in Pharo.*" Indeed, multitasking is prevalent, and can be disruptive [61, 37]. C02 mentioned in the second weekly survey that "*If I am only sending code to Discord (which is IMHO most common for me), then I find the integration not as helpful.*" C02 describes one of the reasons LightShare was unhelpful as:

1. "*I write my code in Pharo,*"
2. "*I switch to Discord to see what [...] the channel [is] I want to send it to,*"
3. "*I switch to Pharo, select the channel and send,*"
4. "*I switch to Discord and @notify the person I [am] talking with.*"

Indeed, LightShare could automatically suggest a channel and a user mention by monitoring the Discord discussion to find out on which channels the user is active and to whom he is speaking; this would require additional integration with Discord besides simply sending messages.

In our final survey, we asked our respondents about their level of integration preference, see Table 6.5. Participants responded: agree 😄, somewhat agree 🙂, neutral 😐, somewhat disagree 🙁, disagree ☹, or I do not know 🙀. Columns represents answers to the following statements:

- Light Integration: "*I am fine with sending messages from Pharo [using LightShare] and continue discussion in the standard Discord client.*",
- Full Integration: "*I would prefer to have the entire conversation inside Pharo (a full Discord chat service integration)*", and

- No Integration: "*I would prefer to have the entire conversation inside the standard Discord client (just use the standard Discord client!).*"

Participant	Light	Full	No
		Integration	
C02	🙀	🙂	🙂
C03	😄	😐	🙂
C05	😄	😐	😐
C08	😐	🙂	🙁
C09	😄	😄	🙁
C10	😄	☹️	😄
C11	😄	🙂	😐
C18	😄	🙁	😐
C21	🙂	😄	☹️
C22	☹️	😄	🙁

Table 6.5: Participant answers on the level of chat and IDE integration using the following scale: agree 😄, somewhat agree 🙂, neutral 😐, somewhat disagree 🙁, disagree ☹️, I do not know 🙀.

Ten participants responded to the final survey. They were satisfied with the level of integration that LightShare provides (share from the IDE, then resume on Discord), finding overall agreement (Agree: 7, Neutral: 1, Disagree: 1, I do not know: 1, see Table 6.5, the Light Integration column). There was less agreement on a full integration between Pharo IDE and Discord (Agree: 6, Neutral: 2, Disagree: 2, the Full Integration column); a smaller proportion of respondents would prefer to conduct all interactions in the standard Discord client only (Agree: 3, Neutral: 3, Disagree: 4, No Integration column).

Increasing the integration may reduce the level of friction. An argument in favor of a full integration between Pharo and Discord is that it could reduce friction. C02 said "*Normally I prefer to use Discord client directly [. . .] At the same time, if I am discussing something related to [a] code, then jumping between [the] Pharo [IDE] and [the] Discord [chat client] can be rather annoying.*"

C11 mentions that a full chat integration could open new doors on how and what developers share regarding their code and knowledge: "*having all conversations in [the] Pharo [IDE] would be nice[,. . .] we could use Pharo to manipulate these conversations [and. . .] to send messages from objects.*" On other hand, he admits that a full chat client integration requires effort and he is not sure if it is worth doing so: "*[It] would be nice to have everything inside Pharo[. . .], but I don't think there is a strong need [. . . for that].*"

C02 also mentioned that Pharo community has different channels and in some of them developers commonly share code snippets, in other channels they do not do so: "*I don't normally extensively exchange code[,. . . there are] usually lots of discussions and rarely [. . .] code snippets. [. . . It] heavily dependent[s] on the channels[,. . .] e.g., in Roassal and Bloc [channels] it is very common to share code samples compared to Iceberg or development [channels].*"

Similarly, C02, C04, C10, and C15 proposed at the beginning of the study several improvements to simplify message sending. They asked: to remember last used Discord account; to remember last included attachments (and attach or not the same types in the next message); and to include only a selected part of code snippets (as this is what they want to discuss). As C02 puts it: "*just a single click in Pharo would be the easiest*".

Tiarks [158] monitors developer interaction data [108] extracted from the IDE to ease the building of human-readable tutorials. Guzzi *et al.* also monitor the IDE to help developer micro-blogging [63]. Monitoring user interactions therefore increases tool usability and we should consider it in our future work to target participant comments and suggestions presented above.

On the other hand, we observed some friction in the Discord client as well. Two of the examples that we described above (the coordinate example, and the padding example) actually happened in the same channel, at the same time; the discussions were overlapping, and at times hard to distinguish between them. The same participants were at times involved in both. This is common on chat platforms [50]. If the conversations were held directly in the IDE, it would be easier to untangle these conversations based on which artifacts are being discussed.

Finally, one possible reason for the lack of usage of the feature to load objects in the IDE is that it has significant friction: From Discord, the user must click on the "Load in Pharo" link, which opens a webpage where the user has to read instructions, select and copy auto-generated Pharo code, swith to the IDE paste the code in the IDE, and finally execute it. If the IDE was monitoring Discord channels itself, it could load the shared artifacts with a single user action, a behavior similar to the one described by C10: "*I have non-Pharo discussions in Discord, and implementing a full Discord client seems like not-invented-here syndrome ... however if [user] mentions [and...] Pharo content [were...] automatically [linked and...] available in running images, that would be nice (a bit like macOS Airdrops ...)*".

Supporting other channels is easier with a lightweight integration. Participants expressed that GIF artifacts should be possible to use in other communication channels, *e.g.,* blog posts, and documentation. Presumably, this applies to screenshots as well. When we presented LightShare to the community for the first time in November 2017, C11 wrote to us "*it would be cool to have some extension for publishing playground screenshots to Twitter*". Another developer declined participation in the study, as he said that he is "*more interested in a Slack integration as we use it in our company.*" There is clearly an interest in facilitating the sharing to other channels beyond Discord. There is a variety of other channels that developers use [109, 149]. Other channels not mentioned by our participants to which the artifacts could be shared are bug reports [169], and tutorials [158].

This is an argument in favor of the lightweight sharing model: it is much easier to post content on a multitude of channels, and use these channels to continue the interactions, as opposed to actively monitor all of the channels from the IDE.

Summary. We observe a consensus that participants prefer a *light* integration to a fully integrated client. Reasons for the partial integration are that it is less likely to interrupt users when programming. On the contrary, the received improvement suggestions indicate a convergence towards a full integration.

6.6. Personal Reflections

On declined usage. We performed in-situ daily deployment study. It was in-situ study as we observed LightShare in a natural context of developer daily work, but we were not able to keep natural conditions as we contacted participants and asked them to use LightShare. It implied that they agreed to use something they had no previous experience with. There was a learning process involved, exploring how and in which situations participants could use LightShare.

The study challenge was probably the fact that users agreed to participate for curiosity. They were willing to try something new. We think that the tool usage declined as participants tried LightShare at the beginning of the study, but over the time they had no particular tangible incentive and immediate reward. Which is a legitimate reason as by the end of the day, participants' business tasks were what they had to fulfill. On other hand, we are not sure why they agreed to participate and why they lost interest. It would be beneficial to ask for participation reasons in the initial survey and the loss of interest at the final survey.

The loss of interest could be counterbalanced by more participants. Unpleasant factor against the study setup is the fact that Pharo users who actively communicated in public Discord channels are in a minority. We contacted 35 users who discussed regularly. We did not ask more people as the discussion participation declined to couple of messages in the analyzed 3-month period.

In summary, we do not think that the LightShare study declined particularly because of LightShare. We also do not think that the study setup was inappropriate, considering what we had access to. We knew in advance that we would have a small participant group and keeping participants active would be challenging. We therefore advertised LightShare every week of the study period by addressing all comments on daily basis. Maybe we could thinking of different survey questions for every week to make it more entertaining. Maybe we could advertise LightShare by publishing blog posts with different use cases. Maybe we could offer a *best user award* for every week, considering a most interesting LightShare usage. On other hand, those ideas would contribute to study biases.

In the end, we chose this study setup as we had no access to better study group. We would do it differently today as the dissertation author is part of a team who researches and develops new programming technologies. The team is made up of about ten people who are software developers and researchers. The team seeks for and builds new technologies that support Live Programming. Team members therefore on daily basis plays with experimental solutions that they constantly change following new ideas and use cases. In addition, the team develops industry products using their own Live Programming environment.

In such a team, we would *not* only observe how LightShare is used, but we would develop a new communication solution from scratch. Such a study report would therefore describe the tool evolution during a prolonged period (one or two years). The report would also include tool use cases and personal experiences developing and using it. This is particularly interesting as exploring new development tools and experiences involve constant changes of programming habits and paradigms. The challenge of such a study is the integration with the PhD curriculum, which has a limited time slot in which a dissertation should be ready for defense.

On low sharing activity. In Section 6.4.3, we report that LightShare supports communication acts that involve sharing of a small frequency (2–3%). The consequence is that it is hard to study an infrequent activity. On other hand, we research tools that do not necessarily support the existing communication habits, but encourage communication habit changes. The prior studies of this dissertation show that using inspector and playground was particularly useful and frequent. LightShare was therefore designed to communicate such acts. As we mentioned before, we were unable to deploy object sharing support for this study. The studied LightShare version therefore does not fully support communicating main scenarios observed in the prior studies.

Nevertheless, we report various use cases where LightShare was particularly useful in this chapter. We can improve the LightShare usage impact understanding in follow-up studies. For example, we could find existing conversations where LightShare could be particularly useful. We could also publish a confirmatory survey to receive different user perspectives of what we observed in this study from a broader public, similarly what we did in Chapter 4. Such analysis could also identify conversations that would require additional LightShare improvements and therefore provide better user experience. In summary, we consider this study successful because we obtained various communication scenarios. We also received a feedback that reveals different user expectations to what electronic communication support should or should *not* be about.

6.7. Summary And Implications

Developers sometimes discuss software artifacts in their online conversations. Unfortunately, these artifacts can be complex and hard to describe only with text. We presented LightShare, an approach to share artifacts directly from an Integrated Development Environment (the Pharo IDE), and to a chat platform (Discord). LightShare automatically adds additional information that enhances the readability of the shared artifacts, and allows developers to load those artifacts in the IDE. After sharing the artifact, users are expected to continue the conversation on the chat platform itself.

We conducted a five week long study, in which 21 participants used LightShare. We find out that LightShare was used infrequently: LightShare was used to share software artifacts 104 times during this period. On public channels (visible to ourselves), our participants shared 27% of their artifacts with LightShare; in general, sharing software artifacts is not a very frequent activity.

If sharing is not frequent, it does not mean it is not useful. We found that our participants shared software artifacts with LightShare in four types of scenarios: 1) to increase the readability of the artifact that they share (by including the expected results, and by highlighting data of interest), 2) to show how some tools are used or where useful information is found in the IDE, 3) to provide additional details to help beginners, and 4) to provide additional information when reporting bugs. Finally, another indicator of perceived usefulness is the fact that 6 developers installed and use LightShare from their own initiative, and that one of our participants prototyped an enhancement to the tool (a user interface to discuss in the IDE).

This implies that when users have an appropriate communication support, messages may contain more useful information than intended and initially needed. For example, if a user sends a message "*use the abracadabra method*", then the only outcome for all readers is the reference to the method. If the user also sends an executable code snippet and a screenshot of the used tool (playground and inspector), different readers can enrich themselves with different knowledge. Some will understand how the method is used (without a difficult exploration). Others will notice some details on the shared tool screenshot and may learn something new about their development environment. We therefore conclude that integrated electronic communication is beneficial for effective software development, and observations apply to variety of communication channels and development tools.

Our participants provided us feedback to further improve LightShare. Participants asked for the possibility to share other artifacts, and to share to other services *e.g.,* Twitter. An important feature, that participants emphasized during the study, is able to share the information with minimal effort. In some cases, switching back and forth between Pharo and Discord was perceived as a significant effort.

A way to reduce this friction would be to increase the level of integration between Pharo and Discord; indeed, some participants expected from the beginning of the study an integrated client for reading incoming messages. However, this conflicts with the requirement of increasing the sharing to additional channels. Indeed, participants had different and sometimes conflicting opinions about the level of integration necessary. A minority of participants preferred not to have any IDE integration; most users were satisfied with the current LightShare approach, still, a similar proportion would prefer a complete integration. It seems that only experience with a fully integrated chat in the IDE could clarify these conflicting opinions.

Although further LightShare development and follow-up studies are required, our observations suggest that integrating electronic communication with programming environments foment message fidelity. It goes hand in hand with observations by Guzzi *et al.* who report that developers are willing to share short messages, including explicit code references [63]. However, how communication about Live Programming should look like is the subject of future work. We propose that such communication should integrate development tools in conversations, allowing to edit and execute code, and to explore runtime information. The question is whether the integrated tools should be the same we observed in Chapter 4 or whether such communication integration requires different Live Programming tools. How to solve technical implementation challenges of such integration also remains a question.

Chapter 7

Conclusions

> *Yesterday I was clever so I wanted to change the world. Today I am wise so I am changing myself.*
>
> — RUMI

In this chapter, we summarize the dissertation contributions, discuss our approach, and point to possible future work.

7.1. Dissertation Contributions

This dissertation makes a number of contributions, for which we summarize them structured as developer questions, Live Programming usage, and LightShare.

Developer questions. We studied what questions developers ask during programming change tasks in Chapter 3. We studied it in the context of Live Programming, and we chose Pharo as a mature Live Programming environment that is used in academia and industry. Our first study answers three research questions as follows:

- *"**What developer questions do Pharo developers ask compared with the observations by Sillito** et al.**?**"* We find that Pharo developers asked questions presented in the prior work, varying in the frequency of individual questions. Our participants favored asking questions about runtime compared to the study by Sillito *et al.* The main reason that our participants observed more often runtime information is the fact, that they had more options to access the runtime information. Our participants were able to run examples, write code snippets, and retrieve objects independently of a debugger.

  We also document eight new questions that complement the existing one. Those new questions cover facts that Pharo is a dynamically typed language, involving some extra steps to answer higher questions; and that some participants used test cases (and debuggers) to deal with their tasks.
- *"**What is the complexity of answering developer questions in Pharo considering involved sub-questions?**"* We analyzed sub-questions that are necessary to

answer root-questions, higher-level questions. We observed that questions about expanding focus points are simplest, while the other questions are complex, involving many sub-questions to answer higher-level questions.

- ***"How do Pharo developers use the development tools compared with the observations by Sillito* et al.*?"*** We analyzed whether participants used static or dynamic tools. Static tools are used to observe and navigate in source code. Dynamic tools are used to observe and navigate running applications. We observed that participants used dynamic tools, frequently answering Finding Focus Points (FFP), Understanding a Subgraph (US), and Questions over Groups of Subgraphs (QGS) root questions. They used predominately static tools answering Expanding Focus Points (EFP) root questions.

Live Programming usage. In the subsequent analysis, presented in Chapter 4, we look in detail at how developers use Live Programming features. We report more closely what strategies and tools they use while they deal with programming change tasks. We complement our exploratory study with 17 programming sessions by a survey and on-line programming videos analysis.

- **"Do developers use the Live Programming features of Pharo?"** We find that Live Programming features are used extensively by Pharo users. In particular, our participants spent 5.5 hours (42%) using dynamic tools and 7.5 hours (58%) using static tools, with 5 sessions having 60% or more of the time spent with dynamic tools.
- **"How do developers use Live Programming in Pharo?"** We describe several Live Programming approaches and contrast them against traditional programming techniques. Our overall finding is that simple approaches were favored, and their combinations can be powerful. Our participants used frequently object inspectors, debuggers, and playgrounds. The early feedback, while exploring applications, is a valuable Live Programming feature.
- **"Do developers, other than those we analyzed, behave similarly?"** We answer this research question in two parts. We first find that surveyed Pharo users agree with most of the observations we made. Subsequently, we observe that users of other languages and tools use more limited capabilities due to the limited extent of the tool support, and with the aim of getting feedback early and often.

LightShare. Previously, we report that Live Programming simplifies program comprehension and Pharo developers favor dynamic information and early feedback. Given that most software is larger than a single person can comprehend on their own, the communication about the software among software developers is of vital importance.

We therefore studied how Live Programming is discussed and how it should be supported by development and communication tools. To this extend, we developed LightShare, a tool for easy communication within Pharo, enriching chat messages with an explicit information and software artifacts. We conducted a five-week long study where the most active users were asked to uses LightShare. The study reports how developers used LightShare and their level of satisfaction:

- **"How often was LightShare used during the study?"** We find that LightShare was used relatively infrequently, understanding that it is *not* a negative result as on-line chat discussions do not have to be accompanied with software artifacts frequently. We observed that the users attached software artifacts in about 2 to 3% in the observed sample. However, we think that this is a consequence of the communication support lack and the resulting user habits.

- **"What information is shared using LightShare?"** The most common shared artifacts were code snippets, screen captures, and combinations thereof; animated GIFs and source code packages were rarely shared. We conclude that the shared artifacts improve the message fidelity and messages are easier to comprehend. Shared code snippets and screen captures were particularly useful. One reason is that sharing code snippets and screen captures using LightShare is particularly easy. We think that this is also closest to what users are used to share and read.

  Animated GIFs may be shared less frequent as some participants reported issues doing so. The cause of the problem was not clear to us and we were not able to improve the situation during the study. Source code packages were likely shared less often as some participants thought that the source code should be shared using source code repositories, *e.g.,* GitHub. One participant was concerned whether it was safe to load the shared source code.
- **"What are the scenarios where LightShare was involved?"** LightShare was involved in a variety of scenarios, including increasing readability (early feedback), learning tools, helping beginners, and reporting bugs. LightShare messages contain more contextual information, implying that message readers easily follow discussions.
- **"Are developers satisfied with LightShare?"** Participants agreed that LightShare is useful, particularly to share screen captures.
- **"What do they propose to improve?"** Participants point out a variety of possible improvements. They suggest to integrate LightShare with other Pharo development tools. They would like to share other artifacts too, such as debugger stack traces and objects. Participants had different opinion about the level of LightShare integration. Some of them expected a full chat integration, some of them see no reason for a fully integrated chat.

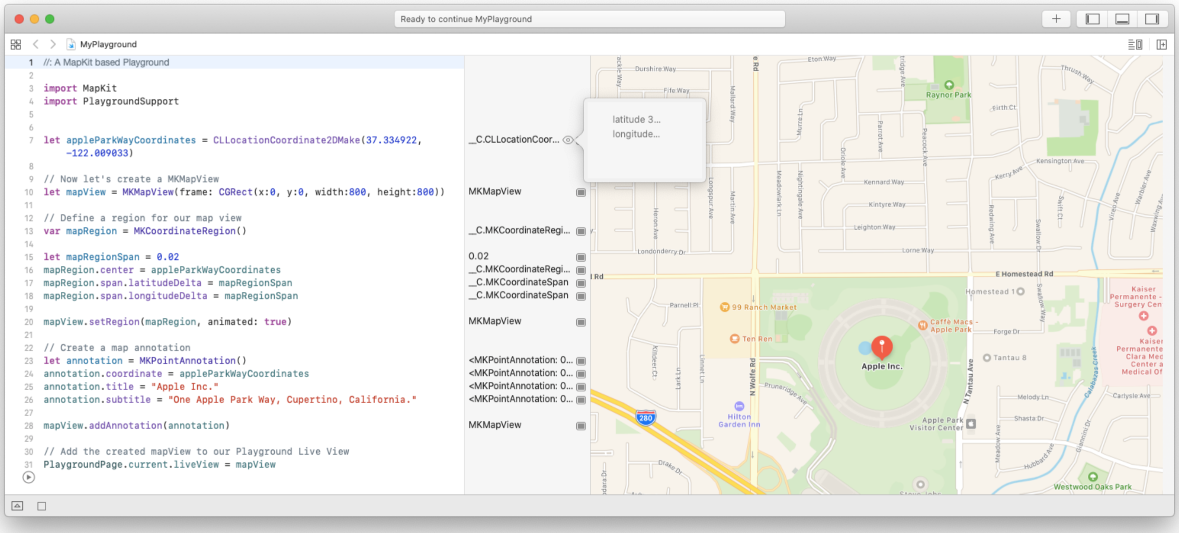
7.2. Live Programming Impact

Live Programming. Questions about software during programming change tasks documented by Sillito *et al.* were also observed in our study. However, our participants addressed questions differently. They preferred to observe runtime information to comprehend software and used different tools and approaches to access the information. They used playground, object inspector, and debugger tools to access the runtime information.

The great advantage of these tools in our study was that they can be used independently. Particularly, it was possible to access objects and display them using object inspectors independently from debuggers. All the tools and information could stay open as long as necessary. We emphasize that this is *not* possible in common development tools, where users usually have only one debugger and one object inspector open at a time.

We conclude that this is an important property of Live Programming. Development tools should be designed as a set of tools that can be used independently and combined as needed. We think that adding a playground and object inspector to integrated development environments could significantly support people during programming tasks.

Recent development tools go in this direction. For example, Figure 7.1 shows Xcode playground. This is a great advance because users can write code snippets and inspect results for individual lines. Some objects have dedicated views. For example, `mapView` is displayed as a map in the figure. On other hand, the object inspector still needs to be improved. The object inspector is tied to playground and can reveal only one piece of information. Users cannot see and change internal details of the `mapView` variable. Which is what we observed in our study: our participants modified objects from object inspectors to understand software behavior.



```
//: A MapKit based Playground

import MapKit
import PlaygroundSupport


let appleParkWayCoordinates = CLLocationCoordinate2DMake(37.334922,
    -122.009033)

// Now let's create a MKMapView
let mapView = MKMapView(frame: CGRect(x:0, y:0, width:800, height:800))

// Define a region for our map view
var mapRegion = MKCoordinateRegion()

let mapRegionSpan = 0.02
mapRegion.center = appleParkWayCoordinates
mapRegion.span.latitudeDelta = mapRegionSpan
mapRegion.span.longitudeDelta = mapRegionSpan

mapView.setRegion(mapRegion, animated: true)

// Create a map annotation
let annotation = MKPointAnnotation()
annotation.coordinate = appleParkWayCoordinates
annotation.title = "Apple Inc."
annotation.subtitle = "One Apple Park Way, Cupertino, California."

mapView.addAnnotation(annotation)

// Add the created mapView to our Playground Live View
PlaygroundPage.current.liveView = mapView
```

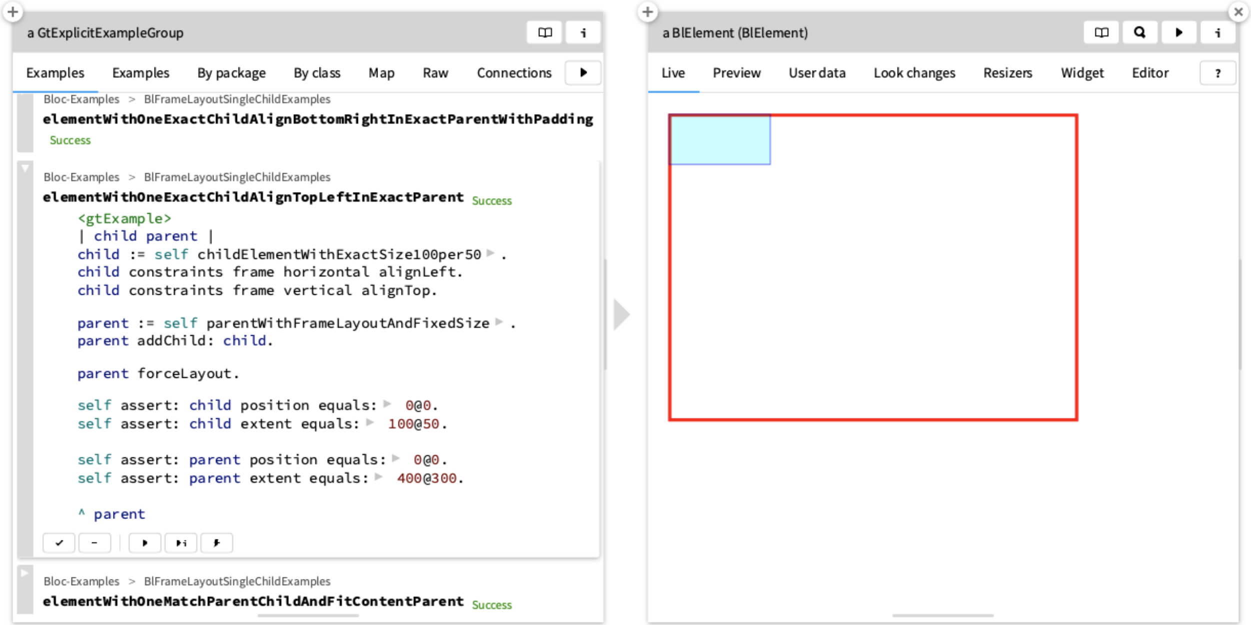
```
__C.CLLocationCoor...

MKMapView

__C.MKCoordinateRegi...

0.02
__C.MKCoordinateRegi...
__C.MKCoordinateSpan
__C.MKCoordinateSpan

MKMapView

<MKPointAnnotation: 0...
<MKPointAnnotation: 0...
<MKPointAnnotation: 0...
<MKPointAnnotation: 0...

MKMapView
```

Figure 7.1: Xcode 11.3 playground with Swift code snippet.

Live examples. Our participants used examples that accompanied software. Examples made it easier to understand software. Some software projects implemented dedicated classes with source code that was possible to execute. Other projects used elaborate example browsers as we described in Section 4.4.2.

We think that development environments should provide a uniform tool to easily write and execute examples. The example tool should be more than just *a documentation tool*, forcing developers to write even more source code. Both developers and users should benefit from such a tool and accompanied examples. It should be therefore favorable to write examples.

The GToolkit programming environment provides such a solution [28]. Figure 7.2 shows an example browser in which examples can be executed and also inspected. Examples are written in a similar way to writing test cases—they include a code that is executed and subsequently tested using assertions. From the developer point of view, he or she writes “tests” and also writes examples for users at the same time. The same source code! No extra source code! No extra work! Both developers and users can inspect and explore example results. As results are inspected using object inspectors, the objects can be further explored and modified. In addition, different objects can have different representations as we can see in Figure 7.2. In this particular case, the inspected object is a graphical widget for which we see its visual and live representation.



Bloc-Examples > BlFrameLayoutSingleChildExamples
elementWithOneExactChildAlignBottomRightInExactParentWithPadding
Success

Bloc-Examples > BlFrameLayoutSingleChildExamples
elementWithOneExactChildAlignTopLeftInExactParent Success

```
<gtExample>
| child parent |
child := self childElementWithExactSize100per50 .
child constraints frame horizontal alignLeft.
child constraints frame vertical alignTop.

parent := self parentWithFrameLayoutAndFixedSize .
parent addChild: child.

parent forceLayout.

self assert: child position equals: 0@0.
self assert: child extent equals: 100@50.

self assert: parent position equals: 0@0.
self assert: parent extent equals: 400@300.

^ parent
```

Bloc-Examples > BlFrameLayoutSingleChildExamples
elementWithOneMatchParentChildAndFitContentParent Success

Figure 7.2: GToolkit example tool displaying an example execution result on the right.

Live communication. We observed that LightShare promoted message fidelity. LightShare was involved in a variety of scenarios, including increasing readability (early feedback), learning tools, helping beginners, and reporting bugs. We conclude that integrating electronic communication with development tools is beneficial to both message senders and message readers. It implies that development tools should support electronic communication during programming change tasks, including Live Programming environments.

LightShare currently support level 3 of message liveness, by including more accurate information. LightShare partially supports (or rather imitates) level 4 of message liveness through the ability to load shared code. Specific conclusions about the level 4 support is a part of future work and we mention possible paths in the following section.

Summary. The objective of this dissertation work is to study Live Programming and electronic communication about Live Programming in practice. The objective is summarized in the following thesis statement: "*We believe that Live Programming is an important property that affects how software developers work and communicate during programming change tasks, and in this dissertation, we study this phenomenon.*"

We conclude that Live Programming provides benefits to software developers and is therefore an important property and positively affects how software developers work. We also affirm that supporting communication about Live Programming is also favorable and the communication should be supported in Live Programming environments. However, the study of Live Programming and communication about it is not complete. The next section provides several directions for the future work.

7.3. Future Work

This dissertation provides incentives to follow-up studies, which we formulate in this section. We propose an additional exploratory session analysis, discuss implementation practices that do and do *not* combine well with Live Programming, propose electronic communication through live documentations, and introduce a new Live Programming environment that emerged in recent years.

Some of the work is already publicly available, although it has not yet been published as a research contribution. For this reason, we mention that some products are already available, although this is the future work section. We include it in this section particularly because the research evaluation and publication are missing.

False assumptions and missed opportunities. In this dissertation, we document what questions developers ask about source code while changing source code. We also report what development tools they use and how they use the tools. Our 17 programming sessions are rich with other information that can be analyzed from different perspectives. For example, we observe about 40 episodes where participants made false assumptions about observed code or missed an information they sought for. It would certainly be valuable to document those episodes and propose development tool improvements the way users make less false assumptions and miss less opportunities.

Live implementation practices. We think that development habits influence the way a source code information can be communicated with higher fidelity, using level 3 or 4 of message liveness. For example, consider a situation where a user wants to understand a squarified treemap algorithm implementation presented by Bruls *et al.* [11]. The squarified treemap algorithm splits an area into sub-areas. Figure 7.3 shows a solution for an area of size 6x4 with 6, 6, 4, 3, 2, 2, and 1 sub-areas.

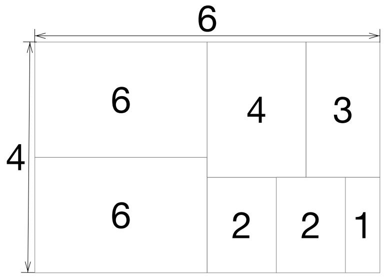


Figure 7.3: An area of 6x4 size divided into seven areas using the squarified treemap algorithm.

It is common to implement the algorithm the way we can observe in the JavaScript D3 project implementation [27]:

```
while (i0 < n) {
  dx = x1 - x0, dy = y1 - y0;
```

```
    // Find the next non-empty node.
    do sumValue = nodes[i1++].value; while (!sumValue && i1 < n);
    minValue = maxValue = sumValue;
    alpha = Math.max(dy / dx, dx / dy) / (value * ratio);
    beta = sumValue * sumValue * alpha;
    minRatio = Math.max(maxValue / beta, beta / minValue);

    // Keep adding nodes while the aspect ratio maintains or improves.
    for (; i1 < n; ++i1) {
      sumValue += nodeValue = nodes[i1].value;
      if (nodeValue < minValue) minValue = nodeValue;
      if (nodeValue > maxValue) maxValue = nodeValue;
      beta = sumValue * sumValue * alpha;
      newRatio = Math.max(maxValue / beta, beta / minValue);
      if (newRatio > minRatio) { sumValue -= nodeValue; break; }
      minRatio = newRatio;
    }

    // Position and record the row orientation.
    rows.push(row = {value: sumValue, dice: dx < dy, children: nodes.slice(i0, i1)});
    if (row.dice) treemapDice(row, x0, y0, x1, value ? y0 += dy * sumValue / value : y1);
    else treemapSlice(row, x0, y0, value ? x0 += dx * sumValue / value : x1, y1);
    value -= sumValue, i0 = i1;
  }
```

It is an excerpt of a full implementation. It takes an effort to understand how it works, which is something that most developers may not care about, but if someone wants to learn, improve, or fix the implementation, the algorithm comprehension becomes vital. For example, we found a squarified treemap implementation in Pharo [29] that for the same data as in Figure 7.3, generates an output as presented in Figure 7.4.

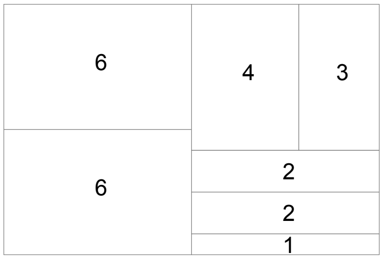


Figure 7.4: A less optimal division of the 6x4 area into seven areas using the squarified treemap algorithm.

The output is less optimal. Figure 7.4 shows a result in which 2, 2, and 1 areas have the heights that are too narrow, compared to their widths. Figure 7.3 shows a solution in which 2, 2, and 1 areas are much closer to square-like representations—their heights and widths are similar and therefore easier to compare with the human eye. We observed the Pharo source code and there are indications that the implementation is inspired by the presented D3 JavaScript source code.

Consider a user who wants to understand the algorithm (and fix it afterwards). It is unlikely that the user wants to learn the algorithm reading the mentioned JavaScript or Pharo source code. It is actually a better idea to read the work by Bruls *et al.* [11] who describes the algorithm as follows, accompanied with Figure 7.5:

> “*The first step of our algorithm is to split the initial rectangle. We choose for a horizontal subdivision, because the original rectangle is wider than high. We next*

fill the left half. First we add a single rectangle (Figure 7.5, step 1). The aspect ratio of this first rectangle is 8/3. Next we add a second rectangle, below the first. The aspect ratios improve to 3/2. However, if we add the next (area 4) below these original rectangles, the aspect ratio of this rectangle is 4/1. Therefore, we decide that we have reached an optimum for the left half in step two, and start processing the right half."

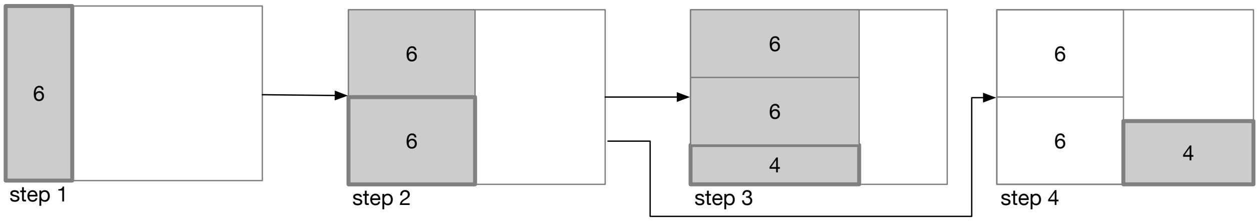


Figure 7.5: Squarified treemap algorithm explanation.

This is much easier to understand yet it still takes an effort to find this description in the mentioned JavaScript code. We think that having a Live Programming development environment to learn about the squarified treemap algorithm implementation does *not* have many benefits. Users would use a source code editor to read the code and a debugger to observe variable value changes. Users of live and non-live development environments would behave similarly.

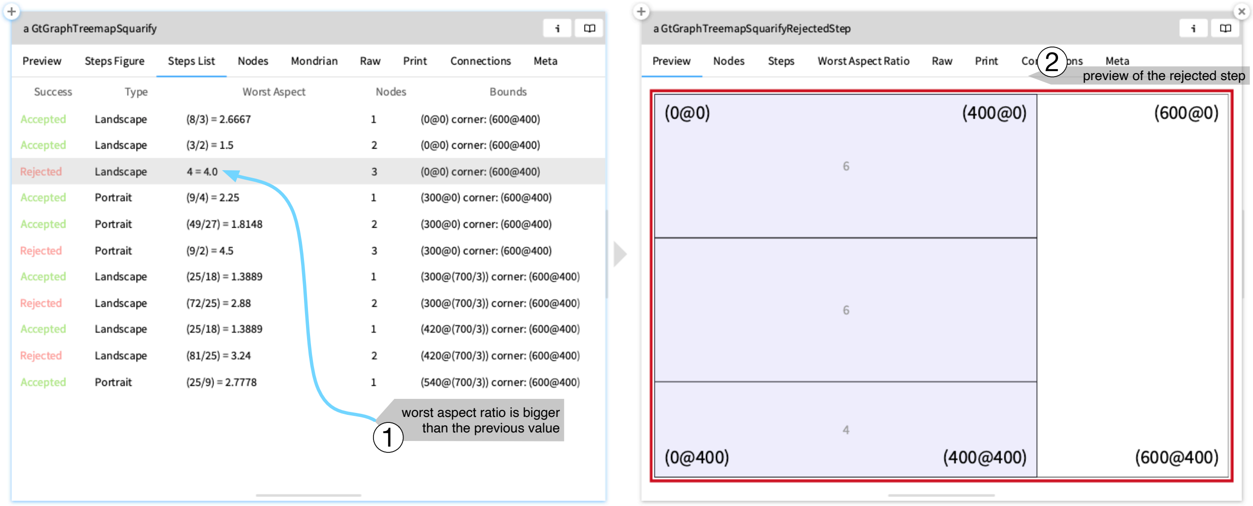
We therefore argue that implementation techniques affect the way developers can benefit from Live Programming development environments. Our future work is to research implementation techniques in which users can take advantage of Live Programming.

Live documentation. We also think that development practices affect the way a source code can be communicated. The communication among developers starts already on the level of source code. The provided JavaScript D3 source code is definitely *not* a documentation that someone wants to learn from. The communication can be improved by an accompanied source code comment, similar to the text extract by Bruls *et al.* Yet, this is not enough as Figure 7.5 is the essential guide.

Consider a squarified treemap implementation that allows one to communicate algorithm details as Figure 7.6 shows. Users can see implementation details, including aspect ratios (part 1) and graphical representations (part 2) for every step. Users also see what steps are rejected and how the final layout presented in Figure 7.3 is constructed.

If we combine this implementation practice with the emerging documentation solutions like Observable [9] and Jupyter [129] notebooks (among others), we just *invented* a new way to communicate source code to users. Figure 7.7 shows an excerpt of such documentation. The documentation embedded *figures* are actually live objects displayed by a moldable inspector [19]. Those objects can therefore be further explored and manipulated while being inside of a class comment.

We already provide a solution that allows us to talk and write about software utilizing *live documentation.* We call it Documenter [94] and it is a part of GToolkit, a Live Programming development environment [28]. Documenter, the live documentation, embeds executable



Success	Type	Worst Aspect	Nodes	Bounds
Accepted	Landscape	(8/3) = 2.6667	1	(0@0) corner: (600@400)
Accepted	Landscape	(3/2) = 1.5	2	(0@0) corner: (600@400)
Rejected	Landscape	4 = 4.0	3	(0@0) corner: (600@400)
Accepted	Portrait	(9/4) = 2.25	1	(300@0) corner: (600@400)
Accepted	Portrait	(49/27) = 1.8148	2	(300@0) corner: (600@400)
Rejected	Portrait	(9/2) = 4.5	3	(300@0) corner: (600@400)
Accepted	Landscape	(25/18) = 1.3889	1	(300@(700/3)) corner: (600@400)
Rejected	Landscape	(72/25) = 2.88	2	(300@(700/3)) corner: (600@400)
Accepted	Landscape	(25/18) = 1.3889	1	(420@(700/3)) corner: (600@400)
Rejected	Landscape	(81/25) = 3.24	2	(420@(700/3)) corner: (600@400)
Accepted	Portrait	(25/9) = 2.7778	1	(540@(700/3)) corner: (600@400)

Figure 7.6: Squarified treemap algorithm implementation details.

examples and executable code snippets whose results can be further explored and manipulated as Live Programming users would expect. Documenter can be used to write class and method comments. Documenter can be also used to write external documents stored for example as individual files.

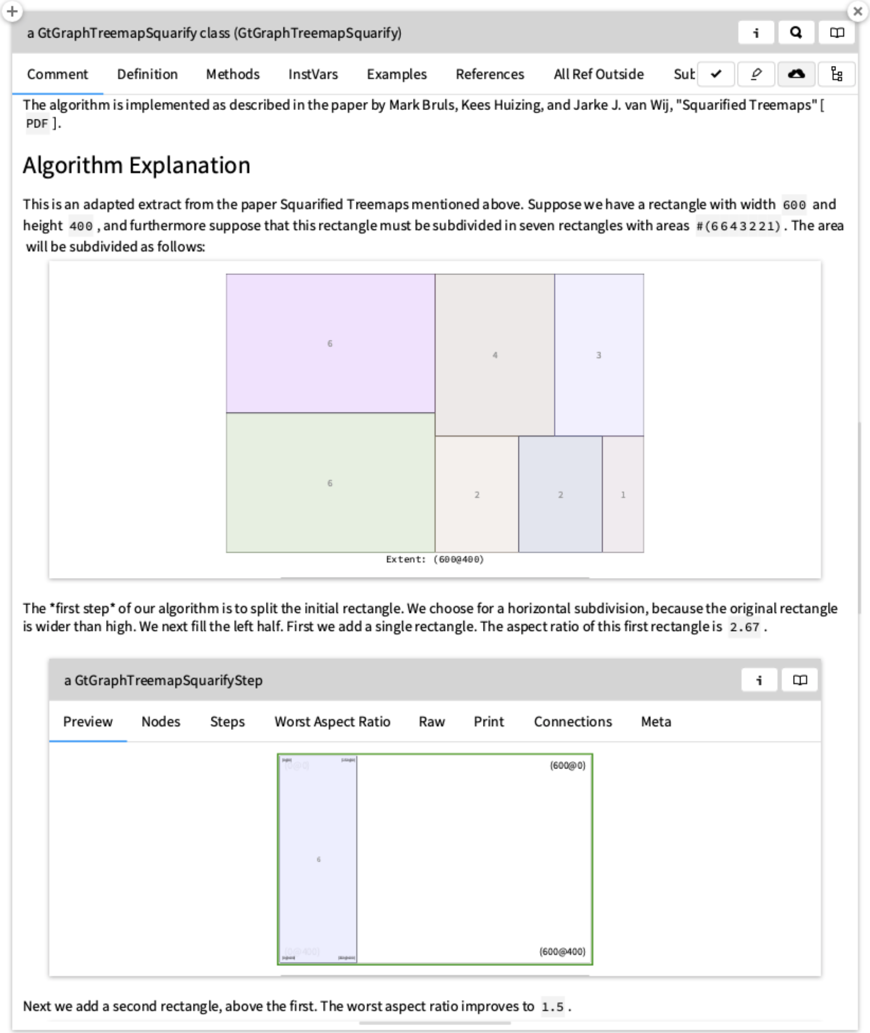
We also conducted a pilot study evaluating Documenter. Our future work is therefore to complete the evaluation and research how individual implementation techniques can be documented and communicated using Documenter in Live Programming environments.

Objects as tools. We just described that implementation techniques influence the way source code can be communicated and documented. We argue that with the right combination of implementation techniques and tool support, live objects can be used in software as an implementation suggest, and at the same time the same live objects can be understood and used as development tools.

For further explanation, consider the following use case. We have a `Resources` object that can search a file (resource) in different places, for example in `/usr/share/`, `/usr/local/share`, and `/home/user` directories. As we know that those places are fixed and will not change in the future, we could choose the following implementation:

```
Resources class >> resourceAtPath: aPath
        | aFileReference |
        "Search in /user/share"
        aFileReference := '/usr/share' asFileReference resolvePath: aPath.
        aFileReference exists ifTrue: [ ^ aFileReference ].
        "Search in /user/local/share"
        aFileReference := '/usr/local/share' asFileReference resolvePath: aPath.
        aFileReference exists ifTrue: [ ^ aFileReference ].
        "Search in /home/user"
        aFileReference := FileLocator home asFileReference resolvePath: aPath.
        aFileReference exists ifTrue: [ ^ aFileReference ].
        self error: 'cannot find resource: ', aPath asString.
```

This is a Pharo code. Particularly it defines a `resourceAtPath:` method with the `aPath` input variable. The method is defined on the class-side of the `Resources` class. The method



The algorithm is implemented as described in the paper by Mark Bruls, Kees Huizing, and Jarke J. van Wij, "Squarified Treemaps" [PDF].

Algorithm Explanation

This is an adapted extract from the paper Squarified Treemaps mentioned above. Suppose we have a rectangle with width `600` and height `400`, and furthermore suppose that this rectangle must be subdivided in seven rectangles with areas `#(6 6 4 3 2 2 1)`. The area will be subdivided as follows:

The *first step* of our algorithm is to split the initial rectangle. We choose for a horizontal subdivision, because the original rectangle is wider than high. We next fill the left half. First we add a single rectangle. The aspect ratio of this first rectangle is `2.67`.

Next we add a second rectangle, above the first. The worst aspect ratio improves to `1.5`.

Figure 7.7: Documentation of the squarified treemap algorithm.

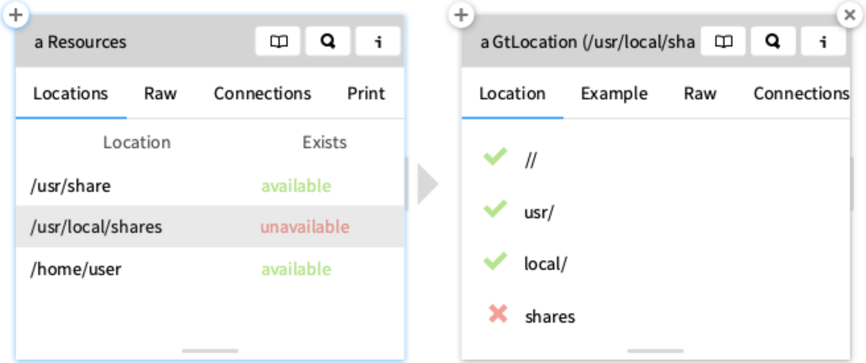
definition uses the following syntax: `| aFileReference |` is a variable definition, `"comment"` is a code comment, `:=` is a variable assignment, and `^ aFileReference` is a return command that returns a `aFileReference` variable value.

The method code looks for a given resource (`aPath`) in `/usr/share/`, `/usr/local/share`, and `/home/user` directories. If it does not find the resource, it raises an error (exception). For clarity, we intentionally introduce the code in this form. An experienced developer would suggest a modular solution.

Now consider a situation where a developer uses the method and receives the error message “cannot find resource” instead of an expected file reference. The problem may be in several places: (i) the `aPath` input variable has incorrect value, (ii) one of the method source code lines do not work as expected, (iii) there are no sufficient permissions to access the resource, (iv) the resource does not exist.

Most users would probably add a breakpoint to the method and use a debugger. They would proceed the code line by line and observe individual variable values. Likely all software developers are familiar with this technique to explore runtime information, regardless of a used programming environment. It has a little to do with Live Programming. Can the source code owner better support users in such scenarios? We think it is possible.

Consider a solution in which the inspected `Resource` object looks as presented in Figure 7.8. Users can immediately spot that there is a problem with the second location, without reading the source code. In this particular case, we deliberately introduced an error in the path definition, `shares` instead of `share`. Readers can imagine that there might be a problem with access rights.



Location	Exists
/usr/share	available
/usr/local/shares	unavailable
/home/user	available

✓ //
✓ usr/
✓ local/
✗ shares

Figure 7.8: Resource object with available and unavailable locations.

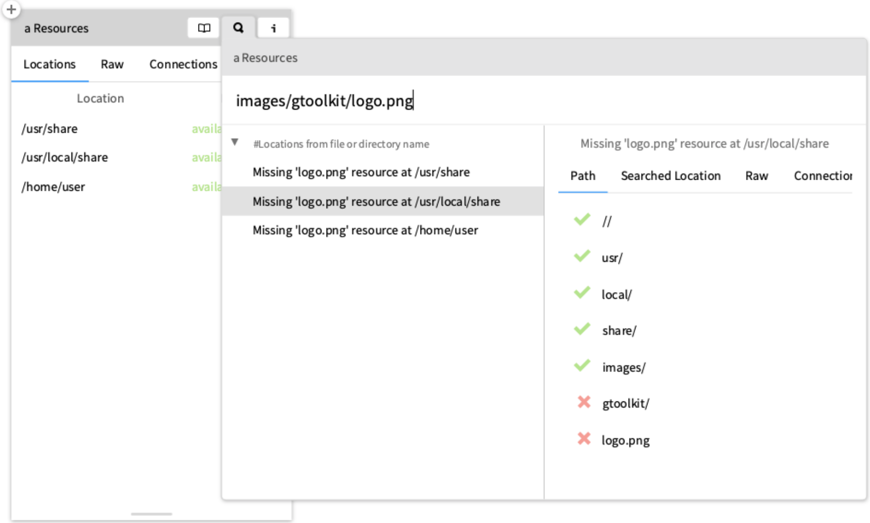
Note that in the previous example, it was not necessary to use the debugger and step through each source code line, observing variable values. Nor was it necessary to read the source code! The information was immediately accessible. Moreover, this object preview is easy to create for individual objects. We used a moldable object inspector to do so [19].

But can we as software developers go further in the resource example? What if all locations are available and a user still receives an error instead of a file reference? Consider that the user searches for `images/gtoolkit/logo.png` file. Ideally, the resource object could have a search dialog as displayed in Figure 7.9.

Figure 7.9 shows that the `images/gtoolkit/logo.png` resource is not available anywhere. In the search dialog preview, he or she can see that `gtoolkit` directory does not exist. After inspecting the search result as shown in Figure 7.10, the user notices that there is a `logos` directory with a `gtoolkit.gt` file instead. The user now knows that the right resource path is `images/logos/gtoolkit.png`.

It is worth noticing that the user just found the problem without leaving his or her development environment. Moreover, without leaving the `Resources` object and without reading the source code! We think that this is a development environment property that belongs to Live Programming and every software developer should prosper by using it.

The presented search extension is actually easy to add. We used Spotter, a moldable search engine [89, 152]. Users can use Spotter to define what and how to search for object information. Summing up this use case story, we described how a few extensions made a regular object (implementation) useful to users. This particular object actually became part of the development environment. Users can either use it to deal with unexpected applica-



Location	
/usr/share	availa
/usr/local/share	availa
/home/user	availa

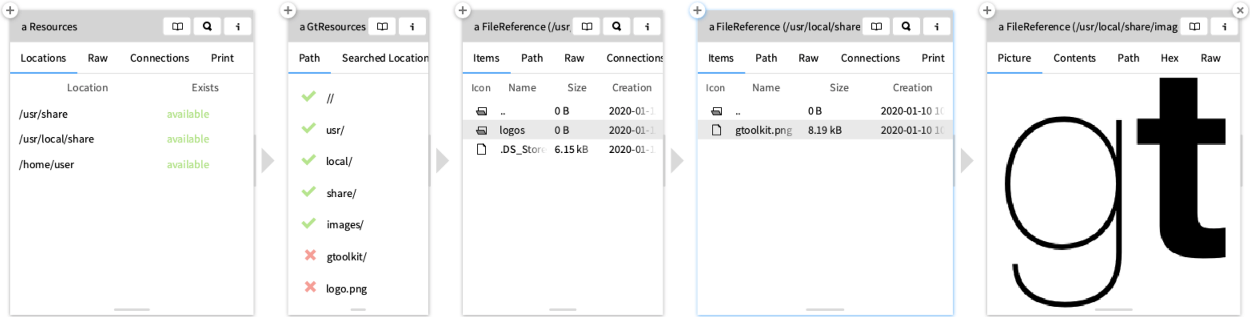
images/gtoolkit/logo.png

#Locations from file or directory name
Missing 'logo.png' resource at /usr/share
Missing 'logo.png' resource at /usr/local/share
Missing 'logo.png' resource at /home/user

Missing 'logo.png' resource at /usr/local/share

✓ //
✓ usr/
✓ local/
✓ share/
✓ images/
✗ gtoolkit/
✗ logo.png

Figure 7.9: Resource object with a Spotter searcher extension.



Location	Exists
/usr/share	available
/usr/local/share	available
/home/user	available

✓ //
✓ usr/
✓ local/
✓ share/
✓ images/
✗ gtoolkit/
✗ logo.png

Icon	Name	Size	Creation
	..	0 B	2020-01-1
	logos	0 B	2020-01-1
	.DS_Store	6.15 kB	2020-01-1

Icon	Name	Size	Creation
	..	0 B	2020-01-10 1(
	gtoolkit.png	8.19 kB	2020-01-10 1(

Figure 7.10: Resource object and a revealed correct resource path.

tion behaviors, and also to search and navigate available resources. The object became a development tool.

Our future work is to research scenarios in which objects can be understood as development environment tools. In the resource example, we achieved it with help of the moldable object inspector and the moldable Spotter search engine. Further tool support is likely to be needed to achieve a similar effect in more variety of cases.

Electronic communication. LightShare is an initial research work exploring how to expose Live Programming in electronic developer communication. The presented solution supports initiating communication from playgrounds. LightShare currently sustains the level 3 of message liveness, and partially support the level 4 of message liveness as users can load a shared code. Our future work is to research how we can reach a full level 4 of message liveness in a way that users will be confident loading and executing received code to their programming environments.

Another open research question is also whether the level 4 of message liveness should look like embedded tools inside of a chat or if it should look like differently. Our assumption is that it should look as close as possible to regular tools. We anticipate that in such conditions users will see and use tools they are used to, and there will be therefore no difference between programming and discussion.

We think that the research on implementation techniques in Live Programming environments and the Documenter usage experience will bring fundamental knowledge about how the electronic communication should look like and be supported. We are confident that integrating Documenter with different communication services and mediums will provide new communication experiences while keeping benefits of Live Programming.

We can already export Documenter documents to various formats, *e.g.,* HTML, Markdown, and Confluence formats. Those formats can be used in standardized communication services like blogs, wiki sites, and issue trackers. Different formats can be also used to communicate with all variety of people who are involved in software development, *e.g.,* chief executive officers, managers, product designers, and marketing persons.

For this purpose we developed XDoc, an *executable document* file format, that holds a variety of live documents and data. XDoc files can be published and shared using standard communication media. Recipients can execute XDoc files using the GToolkit Live Programming environment.

Live Programming environments. We believe that people who are involved in software development should reason about their software products using it from the beginning of development. We mean not only software developers, but also chief executive officers, managers, product designers, marketing persons, and researchers should be able to *play* with their products regardless of the development stage. To efficiently discuss and take *right* decisions about software development, all persons involved should use their products under a discussion at any development phase.

Deploying a *meaningful* software application under development (from now *product*) in every development phase is challenging. In particular, at early development times when there is no graphical interface yet. Often, the product is executable only in development environments. It implies that whomever wants to play with the product must use contemporary integrated development environments. It implies that text editors cannot be a central part of integrated development environments anymore, as it has been for decades. We cannot expect that everyone take a programming course to be able to read, compile, and execute the product source code. Instead, we believe that running products (live objects) should be the central part of development environments. We think that this is what Live Programming stands for.

Such development environments also need to be adaptable to all variety of contexts in order to *display* running applications as expected. For example, a photo editing product should look like a photo editor, and a PDF reader product should look like PDF reader. The current experience is that both the photo editor and the PDF reader look like a list of variable values, until their graphical interfaces are complete and executable. But a lot of source code is implemented before actual graphical interfaces. We argue that Live Programming means that everybody interested and involved in the product development should be able to *play* with the product *from the first line of code.*

GToolkit [28] is a such Live Programming development environment and the dissertation author is a team member of the development team. GToolkit is an industry and research product developed by `feenk.com`. It is implemented with the *moldable tools* concept invented during a PhD work by Andrei Chiş [20]. GToolkit also includes Spotter, Documenter and XDoc mentioned in this section. We also explore new implementation practices that enable one to play with and explore all implementation details. For example, Figure 7.6 represents the squarified treemap algorithm implementation details and Figure 7.7 shows the implementation documentation. Both are part of GToolkit. Our future research work will evaluate GToolkit under different usage scenarios.

Summary. There are many ways to improve Live Programming experience. We highlighted the importance of appropriate development tool support. We also pointed out that tools are not the only requirement and that the implementation techniques need to be changed to take a full advantage of Live Programming. We guided you through two use cases: the squarified treemap and resources implementation. Those examples suggest that a combination of right tools and implementation techniques offers a better user experience and easily converts implementation details to self-explainable solutions and development tools.

We would like to conclude this dissertation with a quote by John Culkin [38]: "*We shape our tools, and thereafter our tools shape us.*", which Tudor Gîrba [58] often completes with a message: "*We should therefore carefully choose our tools*". *We think that the statement can be extended as follows: We should therefore carefully choose our tools and programming techniques.*

Bibliography

[1] Alberto Bacchelli, M. Lanza, and V. Humpa. RTFM (Read the Factual Mails) - Augmenting Program Comprehension with Remail. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 15–24, March 2011.

[2] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-Mails and Source Code Artifacts. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 375–384, New York, NY, USA, 2010. ACM.

[3] Laura Beckwith, Margaret Burnett, Susan Wiedenbeck, Curtis Cook, Shraddha Sorte, and Michelle Hastings. Effectiveness of end-user debugging software features: Are there gender issues? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 869–878. ACM, 2005.

[4] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 231–240. ACM, 2006.

[5] Andrew Begel, Robert DeLine, and Thomas Zimmermann. Social Media for Software Engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 33–38, New York, NY, USA, 2010. ACM.

[6] Alexandre Bergel. *Agile Visualization.* LULU Press, 2016.

[7] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, 2009.

[8] Mike Bland. Testing on the Toilet, October 2011. Available at `https://mike-bland.com/2011/10/25/testing-on-the-toilet.html`.

[9] Mike Bostock and Melody Meckfessel. Observable, a magic notebook for exploring data and thinking with code. Available at `https://observablehq.com/about`.

[10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*,

CSCW '10, pages 301–310, New York, NY, USA, 2010. ACM.

[11] Mark Bruls, Kees Huizing, and Jarke J. van Wijk. Squarified treemaps. In Willem Cornelis de Leeuw and Robert van Liere, editors, *Data Visualization 2000*, pages 33–42, Vienna, 2000. Springer Vienna.

[12] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in UI programming. In *ACM SIGPLAN Notices*, volume 48, pages 95–104. ACM, 2013.

[13] Brian Burg, Adrian Kuhn, and Chris Parnin, editors. *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*. IEEE Computer Society, 2013.

[14] Margaret M Burnett, John Wesley Atwood, and Zachary T Welch. Implementing level 4 liveness in declarative visual programming languages. In *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, pages 126–133. IEEE, 1998.

[15] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 9:1–9:2, New York, NY, USA, 2008. ACM.

[16] Miguel Campusano, Johan Fabry, and Alexandre Bergel. Live Programming in Practice: a Controlled Experiment on State Machines for Robotic Behaviors. *Information and Software Technology*, 108:99 – 114, 2019.

[17] Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. Moldable, context-aware searching with Spotter. In *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, pages 128–144, 2016.

[18] Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. Moldable Tools for Object-Oriented Development. In *Present and Ulterior Software Engineering*, pages 77–101. 2017.

[19] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. The Moldable Inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[20] Andrei Chiş. *Moldable Tools*. PhD thesis, University of Bern, September 2016.

[21] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. *The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers*, pages 102–121. Springer International Publishing, Cham, 2014.

[22] Luke Church, Jonathan Edwards, and Roly Perera, editors. *Proceedings of the 5th International Workshop on Live Programming, LIVE 2019, Athens, Greece, October,*

2019, August 2019.

[23] Discord Collaborators and Users. Discord GitHub Issue Tracker: A Recording GIFs Issue Number 32. Available at https://github.com/JurajKubelka/DiscordSt/issues/32.

[24] Discord Collaborators and Users. Discord GitHub Issue Tracker: An Annimated GIF to Twitter Integration Proposal, Issue Number 31. Available at https://github.com/JurajKubelka/DiscordSt/issues/31.

[25] Code Connect. Code Connect is joining Microsoft, 2016. Available at http://comealive.io/Code-Connect-Joins-Microsoft/.

[26] Pharo Consortium. Pharo website. Available at https://pharo.org.

[27] D3 Contributors. D3 Squarified Treemap Algorithm Implemenation, August 2019. Available at https://github.com/d3/d3-hierarchy/blob/v1.1.9/src/treemap/squarify.js.

[28] GToolkit Contributors. Glamorous Toolkit: A Moldable Development Environment, December 2019. Available at https://gtoolkit.com.

[29] Roassal Contributors. Roassal Squarified Treemap Implementation in Pharo, December 2019. Available at https://github.com/ObjectProfile/Roassal2/blob/004ab6f3dfd41cf06526c57115ee42cc921e7668/src/Roassal2/RTTreeMapBuilder.class.st.

[30] Various Contributors. Wikipedia article about Blog. Available at https://en.wikipedia.org/wiki/Blog.

[31] Wikipedia Contributors. Convenience sampling. Available at https://en.wikipedia.org/wiki/Convenience_sampling.

[32] Wikipedia Contributors. Eclipse Software Article on Wikipedia. Available at https://en.wikipedia.org/wiki/Eclipse_(software).

[33] Wikipedia Contributors. GNU Debugger Article on Wikipedia. Available at https://en.wikipedia.org/wiki/GNU_Debugger.

[34] Wikipedia Contributors. History of Blogging Article on Wikipedia. Available at https://en.wikipedia.org/wiki/History_of_blogging.

[35] Wikipedia Contributors. Instant messaging Article on Wikipedia. Available at https://en.wikipedia.org/wiki/Instant_messaging.

[36] Wikipedia Contributors. Q&A Software Article on Wikipedia. Available at https://en.wikipedia.org/wiki/Q&A_software.

[37] Luis C. Cruz, Heider Sanchez, Víctor M. González, and Romain Robbes. Work frag-

mentation in developer interaction data. *Journal of Software: Evolution and Process*, 29(3), 2017.

[38] John M. Culkin. A Schoolman's Guide to Marshall McLuhan. *The Saturday Review*, pages 51–53, March 1967. Available at `https://www.unz.com/print/SaturdayRev-1967mar18-00051/`.

[39] Brian de Alwis and Gail C. Murphy. Answering Conceptual Queries with Ferret. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 21–30, New York, NY, USA, 2008. ACM.

[40] Peli de Halleux and Michal Moskal. TouchDevelop Programming Environment. Available at `https://www.microsoft.com/en-us/research/project/touchdevelop/`.

[41] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger Canvas: Industrial experience with the code bubbles paradigm. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 1064–1073. IEEE, 2012.

[42] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James F Terwilliger, and John Wernsing. Tempe: Live scripting for live data. In *VL/HCC*, pages 137–141, 2015.

[43] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Beatriz Arévalo. Fuel: A Fast General Purpose Object Graph Serializer. *Software: Practice and Experience*, June 2012.

[44] O. Díaz and C. Arellano. Integrating microblogging into domain specific language editors. In *2013 International Conference on Cloud and Green Computing*, pages 219–225, Sept 2013.

[45] Discord. The Discord Chat Service Site, June 2019. Available at `https://discordapp.com`.

[46] E. Duala-Ekoko and M.P. 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.

[47] Jonathan Edwards. Subtext: uncovering the simplicity of programming. *ACM SIGPLAN Notices*, 40(10):505–518, 2005.

[48] Lilia Efimova and Jonathan Grudin. Crossing boundaries: A case study of employee blogging. In *40th Hawaii International International Conference on Systems Science (HICSS-40 2007), CD-ROM / Abstracts Proceedings, 3-6 January 2007, Waikoloa, Big Island, HI, USA*, page 86, 2007.

[49] Keith Eliott. Swift Playgrounds—Interactive Awesomeness, 2016. Available at `http://bit.ly/Swift-Playgrounds`.

[50] Micha Elsner and Eugene Charniak. Disentangling Chat. *Computational Linguistics*,

36(3):389–409, 2010.

[51] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How Do API Documentation and Static Typing Affect API Usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 632–642, New York, NY, USA, 2014. ACM.

[52] Johan Fabry. The Meager Validation of Live Programming. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Jhonny Wilder Cerezo Felipez, Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Building an expert recommender chatbot. In *Proceedings of the 1st International Workshop on Bots in Software Engineering, BotSE@ICSE*, pages 59–63, May 2019.

[54] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference.* O'Reilly Media, Inc, 4th edition edition, 2002.

[55] Uwe Flick. *An introduction to qualitative research.* Sage, 2014.

[56] Thomas Fritz and Gail 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*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[58] Tudor Gîrba. Tudor Gîrba's website, December 2019. Available at `http://www.tudorgirba.com`.

[59] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983.

[60] Max Goldman, Greg Little, and Robert C. Miller. Real-time Collaborative Coding in a Web IDE. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 155–164, New York, NY, USA, 2011. ACM.

[61] Victor M. González and Gloria Mark. "Constant, Constant, Multi-tasking Craziness": Managing Multiple Working Spheres. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 113–120, New York, NY, USA, 2004. ACM.

[62] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. Communication in Open Source Software Development Mailing Lists. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 277–286, Piscataway, NJ, USA, 2013. IEEE Press.

[63] Anja Guzzi, Martin Pinzger, and Arie van Deursen. Combining micro-blogging and IDE interactions to support developers in their quests. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5, Sept 2010.

[64] Christopher Michael Hancock. *Real-time programming and the big ideas of computational literacy.* PhD thesis, Massachusetts Institute of Technology, 2003.

[65] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[66] Scott Hanselman. Interactive Coding with C# and F# REPLS, 2016. Available at `http://bit.ly/InteractiveCodingCF`.

[67] Chih-Wei Ho, Somik Raha, Edward Gehringer, and Laurie Williams. Sangam: A Distributed Pair Programming Plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '04, pages 73–77, New York, NY, USA, 2004. ACM.

[68] Christopher D. Hundhausen and Adam S. Carter. Supporting Social Interactions and Awareness in Educational Programming Environments. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 55–56, New York, NY, USA, 2014. ACM.

[69] CollabNet Inc. ArgoUML, an open source UML modeling tool. Available at `http://argouml.tigris.org`.

[70] Crunchbase Inc. VS Anywhere, a Visual Studio extension for a real-time multi-person collaboration platform. Available at `https://www.crunchbase.com/organization/vs-anywhere`.

[71] Eclipse Foundation Inc. Eclipse IDE, June 2019. Available at `https://www.eclipse.org`.

[72] Stack Exchange Inc. Stack Overflow Site. Available at `http://stackoverflow.com`.

[73] Twitter Inc. Twitter website. Available at `https://twitter.com`.

[74] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan C. Kay. Back to the future. *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97*, 1997.

[75] Aditya Johri. Look Ma, No Email!: Blogs and IRC As Primary and Preferred Communication Tools in a Distributed Firm. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 305–308, New York, NY, USA, 2011. ACM.

[76] Aditya Johri. Supporting Global Virtual Work through Blogs and Micro-blogging. In *System Sciences (HICSS), 2015 48th Hawaii International Conference on*, pages 422–

431, Jan 2015.

[77] Natalia Juristo and Sira Vegas. The role of non-exact replications in software engineering experiments. *Empirical Software Engineering*, 16(3):295–324, 2011.

[78] Alan Kay. Squeak Etoys, Children & Learning. Technical Report VPRI Research Note RN-2005-001, Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201, 2005.

[79] Alan C. Kay. A Personal Computer for Children of All Ages. In *Proceedings of the ACM National Conference*. ACM Press, August 1972.

[80] Alan C. Kay. The Early History of Smalltalk. In *ACM SIGPLAN Notices*, volume 28, pages 69–95. ACM Press, March 1993.

[81] Alan C. Kay. Squeak Etoys Authoring & Media. Technical Report VPRI Research Note RN-2005-002, Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201, 2005.

[82] Andrew J. Ko, Htet Htet Aung, Brad A. Myers, and M. J. Coblenz. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

[83] Andrew J. 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.

[84] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1569–1578. ACM, 2009.

[85] W. Kozaczynski, S. Letovsky, and J. Ning. A Knowledge-based Approach To Software System Understanding. In *Knowledge-Based Software Engineering Conference, 1991. Proceedings., 6th Annual*, pages 162–170, Sep 1991.

[86] Juraj Kubelka. Artifact Driven Communication To Improve Program Comprehension. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 457–460, 2017. Presented at Doctoral Symposium.

[87] Juraj Kubelka. LightShare 0.2.0 to 0.2.5 milestones on the GitHub issue tracker, April 2018. Available at `https://github.com/JurajKubelka/DiscordSt/milestones?state=closed`.

[88] Juraj Kubelka, Alexandre Bergel, Andrei Chiş, Tudor Gîrba, Stefan Reichhart, Romain Robbes, and Aliaksei Syrel. EventRecorder: User Interaction Collection Framework for Pharo, May 2015. Available at `https://github.com/pharo-contributions/EventRecorder`. The original code available at `http://smalltalkhub.com`, the Moose

team, the GToolkit project.

[89] Juraj Kubelka, Alexandre Bergel, Andrei Chiş, Tudor Gîrba, Stefan Reichhart, Romain Robbes, and Aliaksei Syrel. On Understanding How Developers Use the Spotter Search Tool. In *Proceedings of 3rd IEEE Working Conference on Software Visualization - New Ideas and Emerging Results*, VISSOFT-NIER'15, pages 1–5. IEEE, 2015.

[90] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and Answering Questions During a Programming Change Task in Pharo Language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014.

[91] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Pitekün, an experimental programming environment for Pharo, July 2014. Available at `https://github.com/JurajKubelka/Pitekun`.

[92] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Pitekün: An Experimental Visual Tool to Assist Code Navigation and Code Understanding. In *Proceedings of Jornadas Chilenas de Computación 2014*, JCC '14, pages 1–4, 2014.

[93] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. LightShare, a Discord client interagted in Pharo, April 2018. Available at `https://github.com/JurajKubelka/DiscordSt`.

[94] Juraj Kubelka, Tudor Gîrba, Ioana Verebi, Aliaksei Syrel, Andrei Chiş, John Brant, George Ganea, Manuel Leuenberger, and Alistair Grant. Documenter: Tool For Creating And Consuming Live Documents Directly In The Development Environment, December 2019. Available at `https://github.com/feenkcom/gtoolkit-documenter`.

[95] Juraj Kubelka, Tudor Gîrba, Ioana Verebi, Aliaksei Syrel, Andrei Chiş, John Brant, George Ganea, Manuel Leuenberger, and Alistair Grant. XDoc: Executable Document File Format for Live Documents, December 2019. Available at `https://github.com/feenkcom/xdoc`.

[96] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. ICSE 2018 and ICPC 2019 Session Video Recordings, February 2018. Available at `https://doi.org/10.5281/zenodo.1170460`.

[97] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. LightShare Study Surveys 2018, May 2018. Available at `https://doi.org/10.5281/zenodo.3603485`.

[98] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The Road to Live Programming: Insights from the Practice. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1090–1101, 2018.

[99] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. ICSE 2018 and ICPC 2019 Research Dataset on Live Programming, March 2019. Available at: `https://doi.org/10.5281/zenodo.2591188`.

[100] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. ICSE 2018 and ICPC 2019 Research Tools, March 2019. Available at `https://doi.org/10.5281/zenodo.2591209`.

[101] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Live Programming and Software Evolution: Questions during a Programming Change Task. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 30–41, 2019.

[102] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program Comprehension As Fact Finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

[103] Thomas D. LaToza and Brad 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.

[104] Thomas D. LaToza and Brad A. Myers. Hard-to-answer Questions About Code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[105] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[106] Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. Why Developers Are Slacking Off: Understanding How Software Teams Use Slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, CSCW '16 Companion, pages 333–336, New York, NY, USA, 2016. ACM.

[107] Jesús Jara López, Lina Bautista, and Iván Paz, editors. *Proceedings of the Fourth International Conference on Live Coding, Madrid, Spain, January, 2019*, 2019.

[108] Walid Maalej, Thomas Fritz, and Romain Robbes. Collecting and processing interaction data for recommendation systems. In *Recommendation Systems in Software Engineering*, pages 173–197. Springer, 2014.

[109] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4):31:1–31:37, September 2014.

[110] John H. Maloney and Randall B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM.

[111] Sean McDirmid. Living it up with a live programming language. In *ACM SIGPLAN*

Notices, volume 42, pages 623–638. ACM, 2007.

[112] Sean McDirmid. The Promise of Live Programming. In *Proceedings of the 2nd International Workshop on Live Programming*, LIVE '16, Rome, Italy, 2016.

[113] Sean McDirmid and Jonathan Edwards. Programming with managed time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 1–10. ACM, 2014.

[114] Sean McDirmid and Wilson C. Hsieh. Superglue: component programming with object-oriented signals. In *ECOOP*. Springer Verlag, June 2006.

[115] Leonel Merino, Mohammad Ghafari, Oscar Nierstrasz, Alexandre Bergel, and Juraj Kubelka. Metavis: Exploring actionable visualization. In Bonita Sharif, Christopher Parnin, and Johan Fabry, editors, *2016 IEEE Working Conference on Software Visualization, VISSOFT 2016, Raleigh, NC, USA, October 3-4, 2016*, pages 151–155. IEEE Computer Society, 2016.

[116] Miguel Campusano and Alexandre Bergel. VizRob: Effective Visualizations to Debug Robotic Behaviors. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 86–93, Feb 2019.

[117] 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.

[118] S. C. Müller and T. Fritz. Stakeholders' Information Needs for Artifacts and Their Dependencies in a Real World Context. In *2013 IEEE International Conference on Software Maintenance*, pages 290–299, Sept 2013.

[119] Emerson Murphy-Hill and Gail C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 405–414, New York, NY, USA, 2011. ACM.

[120] Emerson Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[121] Emerson Murphy-Hill, Edward K. Smith, Caitlin Sadowski, Ciera Jaspan, Collin Winter, Matthew Jorde, Andrea Knight, Andrew Trenk, and Steve Gross. Do developers discover new tools on the toilet? In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 465–475, Piscataway, NJ, USA, 2019. IEEE Press.

[122] Oracle. Java Platform Debugger Architecture, 2014. Available at `http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/`.

[123] Christopher Parnin. A History of Live Programming, 01 2013. Available at `http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/`.

[124] Christopher Parnin, Christoph Treude, and Margaret-Anne Storey. Blogging developer knowledge: Motivations, challenges, and future directions. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 211–214, May 2013.

[125] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 212–222. ACM, 2014.

[126] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering: Theory and Practice.* Pearson, 2006.

[127] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1295–1298, Piscataway, NJ, USA, 2013. IEEE Press.

[128] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Prompter - Turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, pages 1–42, 2015.

[129] M Ragan-Kelley, F Perez, B Granger, T Kluyver, P Ivanov, J Frederic, and M Bussonnier. The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication. In *AGU Fall Meeting Abstracts*, 2014.

[130] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. How Live are Live Programming Systems? *Proceedings of the Programming Experience 2016 (PX/16) Workshop on - PX/16*, 2016.

[131] Martin P Robillard, Wesley Coelho, and Gail C Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903, 2004.

[132] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How Do Professional Developers Comprehend Software? In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[133] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201. ACM, 2015.

[134] Stephan Salinger, Christopher Oezbek, Karl Beecher, and Julia Schenk. Saros: An Eclipse Plug-in for Distributed Party Programming. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 48–55, New York, NY, USA, 2010. ACM.

[135] Erik Sandewall. Programming in an Interactive Environment: The "Lisp" Experience. *ACM Comput. Surv.*, 10(1):35–71, March 1978.

[136] Julia Schenk, Lutz Prechelt, and Stephan Salinger. Distributed-pair Programming Can Work Well and is Not Just Distributed Pair-programming. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 74–83, New York, NY, USA, 2014. ACM.

[137] Till Schümmer and Stephan Lukosch. *Supporting the Social Practices of Distributed Pair Programming*, pages 83–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[138] E. Shihab, Zhen Ming Jiang, and A. E. Hassan. On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK+ project. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 107–110, May 2009.

[139] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 9–19. IEEE, 2015.

[140] Jonathan Sillito. *Asking and answering questions during a programming change task*. PhD thesis, University of British Columbia, Feb 2006.

[141] Jonathan Sillito, Kris De Voider, Brian Fisher, and Gail C. Murphy. Managing software change tasks: an exploratory study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–, Nov 2005.

[142] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[143] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and Answering Questions during a Programming Change Task. *Software Engineering, IEEE Transactions on*, 34(4):434–451, July 2008.

[144] Leif Singer, Fernando Figueira Filho, and Margaret-Anne Storey. Software Engineering at the Speed of Light: How Developers Stay Current Using Twitter. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 211–221, New York, NY, USA, 2014. ACM.

[145] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet, a collaboration system architecture. In *Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing*, pages 2–9, 2003.

[146] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259, New York, NY, USA, 2015. ACM.

[147] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. Wysiwis revised: Early

experiences with multiuser interfaces. *ACM Trans. Inf. Syst.*, 5(2):147–167, April 1987.

[148] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. The (R) Evolution of Social Media in Software Engineering. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 100–116, New York, NY, USA, 2014. ACM.

[149] Margaret-Anne Storey, Alexey Zagalsky, Fernando Marques Figueira Filho, Leif Singer, and Daniel M. Germán. How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development. *IEEE Trans. Software Eng.*, 43(2):185–204, 2017.

[150] Stripe. The Developer Coefficient. Available at `https://stripe.com/files/reports/the-developer-coefficient.pdf`.

[151] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[152] Aliaksei Syrel, Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, and Stefan Reichhart. Spotter: Towards a Unified Search Interface in IDEs. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 54–55, New York, NY, USA, 2015. ACM.

[153] Steven L Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[154] Steven L. Tanimoto. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press.

[155] Google Chrome Development Team. Google Chrome Development Tools, 2017. Available at `https://developers.google.com/web/tools/chrome-devtools/`.

[156] React Development Team. React: A Javascript Library For Building User Interfaces, 2017. Available at `https://facebook.github.io/react/`.

[157] Nelson Tenório and Pernille Bjørn. How a geographically distributed software team managed to negotiate successfully using chat technology. *Revista Tecnologia e Sociedade*, 15(37), July 2019.

[158] Rebecca Tiarks. *How-To Software Knowledge*. PhD thesis, Department of Informatics of the University of Hamburg, München, 2015.

[159] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How Do Programmers Ask and Answer Questions on the Web? (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, New York, NY, USA, 2011. ACM.

[160] Christoph Treude and Margaret-Anne D. Storey. Effective communication of software development knowledge through community portals. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 91–101, 2011.

[161] Kasey Uhlenhuth. Introducing the Microsoft Visual Studio C# REPL, Nov 2015. Available at `https://channel9.msdn.com/Events/Visual-Studio/Connect-event-2015/103`.

[162] D. Ungar and R.B. Smith. SELF: the power of simplicity (object-oriented language). *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*, 1988.

[163] Bret Victor. Inventing on Principle. Invited Talk at CUSEC'12. Video recording available at `http://vimeo.com/36579366`. Accessed: 26-04-2018.

[164] J. Whitehead. Collaboration in Software Engineering: A Roadmap. In *Future of Software Engineering, 2007. FOSE '07*, pages 214–225, May 2007.

[165] Steve Whittaker, Victoria Bellotti, and Paul Moody. Introduction to This Special Issue on Revisiting and Reinventing e-Mail. *Hum.-Comput. Interact.*, 20(1):1–9, June 2005.

[166] EM Wilcox, J William Atwood, Margaret M Burnett, Jonathan J Cadiz, and Curtis R Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 258–265. ACM, 1997.

[167] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, 24(2):637–673, July 2018.

[168] Franz Zieris and Lutz Prechelt. Observations on knowledge transfer of professional software developers during pair programming. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 242–250. IEEE, 2016.

[169] T. Zimmermann, R. Premraj, Nicolas Bettenburg, S. Just, A. Schroter, and C. Weiss. What Makes a Good Bug Report? *Software Engineering, IEEE Transactions on*, 36(5):618–643, Sept 2010.