Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Sources of Software Development Task Friction

Nick C. Bradley[1] · **Thomas Fritz**[2] ·
Reid Holmes[1]

Received: date / Accepted: date

Abstract Given a task description, a developer's job is to alter the software system in a way that accomplishes the task, usually by fixing a bug or adding a new feature. Completing these tasks typically requires developers to use multiple tools, spanning multiple applications, within their environment. In this paper, we investigate how existing desktop environments align with and facilitate developers' needs as they tackle their tasks. We examine how developers use their tools to perform their tasks and the ways in which these tools inhibit development velocity. Through a controlled user study with 17 subjects and a field study with 10 industrial engineers, we found that developers frequently formulate specific objectives, or goals, on-demand as they encounter new information when progressing through their tasks. These goals are often not achievable directly in the environment, forcing developers to translate their task into goals and their goals into the low-level actions provided by the environment. When carrying out these low-level actions, developers routinely perform extra work such as locating and integrating resources and adapting their needs to align with the capabilities of the environment. This extra work acts as a form of friction, limiting how quickly and directly developers can complete their tasks. Much of this extra work exists due to mismatches between current tools and environments and how developers actually work in practice. This work identifies seven types of development friction and provides design recommendations that future tools and environments could use to more effectively help developers complete their tasks.

Keywords Developer tasks · Developer goals · Developer tools · Development friction · Integrated Development Environments · Human aspects of SE

✉ Nick C. Bradley (ncbrad@cs.ubc.ca)
Thomas Fritz (fritz@ifi.uzh.ch)
Reid Holmes (rtholmes@cs.ubc.ca)
[1] The University of British Columbia, Vancouver, Canada
[2] Department of Informatics, University of Zurich, Zurich, Switzerland

1 Introduction

Developers add value to their systems by making changes that fix defects or add new features requested in their tasks. However, developers must first break down the task description into concrete objectives, or goals, which they complete by performing actions across different applications in the environment. These goals motivate developers' actions and represent their intentions as they seek out and respond to feedback from their tools and combine it with their experience and observations. Unfortunately, actions are often spread across multiple applications making it difficult for developers to obtain, integrate, and use their feedback (Maalej, 2009; Ko et al, 2006; Sillito et al, 2008). While prior work has looked at developers' information needs across tasks (Ko et al, 2006; Sillito et al, 2008), at specific activities such as code comprehension (Robillard et al, 2004), debugging (Lawrance et al, 2013), and testing (Beller et al, 2015) and at specific tools, especially the IDE (Minelli et al, 2015a; Amann et al, 2016), little is known about the overall process of developers breaking down their tasks into goals and actions and, in particular, the challenges they face in doing so.

To identify these challenges, we first need to understand what developers are trying to accomplish—their goals. Developers formulate their goals based on the description of the bug, maintenance, or feature task they are assigned. They operationalize their goals by performing the low-level actions provided across the applications in their environment (Wang and Chiew, 2010; Chattopadhyay et al, 2019). In the context of coding, Soloway (1986) describes this process as stepwise refinement where developers decompose their tasks into sterotypical solutions (goals) consisting of fixed sequences of actions. For example, a developer working on a bug fixing task may have the goal of testing their changes. To operationalize this goal, they may have to switch to their shell application and run commands to build and start their program, and then switch to the running program and observe changes in behaviour.

This indirect translation from task descriptions-to-goals and goals-to-actions can be mentally demanding as developers need to ascertain goals and ideate how they can complete those goals with the available tool actions. However, these actions often require developers to locate, integrate, and use specific information across applications. This separation between information and actions requires developers to remember information as they switch between applications and keep track of the windows and resources containing relevant information. As developers complete goals, they must also organize or close resources that are no longer needed to maintain a usable environment. According to Green's Cognitive Dimensions (1989), such a system is considered viscous as it requires developers to perform extra actions across multiple resources. We refer to these extra actions as *friction* because they reduce the velocity at which developers can accomplish their goals. In this paper, we investigate friction through the following research questions.

RQ1 How do developers operationalize goals into low-level actions provided by the environment?
RQ2 What kinds of friction do developers encounter when performing their tasks?

To answer these questions, we conducted two studies in which we collected the screen recordings of 27 developers as they used cross-application workflows to complete their tasks. We observed 17 developers in controlled settings address a real issue on a medium size open-source project and 10 developers in naturalistic settings working on their own tasks. Taking inspiration from the Cognitive Walkthrough method (Lewis and Wharton, 1997), we examined participants' workflows through a goal-centric lens to identify discrepancies between the goals they wanted to accomplish and how they were able to operationalize them as sequences of tool actions. Together, these two studies allowed us to investigate how developers decompose a diverse set of tasks into goals and the frictions they experience operationalizing these goals across applications.

We found that developers often derive similar sequences of goals when decomposing the same task and frequently formulate goals on-demand when they encountered obstacles. Operationalizing these goals to actions is an indirect process that requires developers to perform actions and move information across multiple applications and resources. While operationalizing their goals, developers often encounter friction getting to their resources, integrating information between windows and resources, and adapting the environment to support the completion of their goals.

This paper provides insight into how developers decompose their tasks into goals and how they complete those goals in their environment. We identify three design aspects of the environment which cause developers to experience seven types of friction in their cross-application workflows. Finally, we contribute three design recommendations which seek to mitigate friction at a more fundamental level by better aligning environments with the ways developers want to accomplish their tasks. This work motivates further research into addressing the frictions imposed on developers by environments to enable them to more effectively accomplish their tasks.

2 Definitions

A list of terms frequently used throughout this work are included here to clarify their meanings; these are based in-part on the task framework developed by Byström and Hansen (2005), a summary of which we reproduce here:

> “ We recognize that there are more or less specific *task descriptions*, sometimes in objective terms, but always in subjective terms (that is, as perceived by a task performer), that direct the *task processes*. Additionally, work tasks often consist of subtasks and may themselves be considered as subtasks to larger projects. A work task may be divided into subtasks that must be accomplished and connected in order to reach a meaningful result that is related to the task performer's duties.

We use this framework to disambiguate the various uses of task (e.g., Dragunov et al (2005); Oliver et al (2006); Rattenbury and Canny (2007)) and activity (e.g., Bannon et al (1983); Jeuris et al (2014); Bardram et al (2019)) found in the literature.

Task. In this paper, task refers to both the *task description*, provided either orally or as issues and todo items, and the *task process*, the actions developers perform to reach their goal of a “meaningful result.” We alternately refer to task processes as goal operationalizations or *workflows*.

Activity. A descriptive high-level label summarizing observed task processes using one of several researcher-defined categories. Activities simply aggregate developer actions and do not depend on the developer's current context or goal. For example, the action of setting a breakpoint could be described as a debugging activity without considering the developer's overall task.

Goal. The ends towards which a developer's effort is directed (Chattopadhyay et al, 2019). When trying to understand a developer's work, goals provide the *why* while task descriptions provide the *what* and task processes provide the *how*. Just as tasks can be divided in subtasks, so too can goals be divided into more specific subgoals. For example, given a task description, a developer needs to think about the steps they will take to complete it. This could start with questions: is the task well-specified? what aspects of the system does it affect? what changes would need to be made? The developer chooses one of the questions as their current goal and then considers the actions they could take to answer the question. As they perform the necessary actions, new questions may arise which then become subgoals they must satisfy with another set of actions. These actions cause changes to the resources in the environment which culminate in the completion of the task and constitute the task process or workflow.

Resource. Any object represented in the environment that enables developers to record and communicate information to themselves and other developers. Common resources include files, documentation, and issues. We also consider shell commands and tool panes within applications to be resources as they communicate information to developers. For example, the file and search panes in IDEs provide information about the structure and content of projects while shell commands like `git status` provide information about changes to the project.

Action. Any operation a developer can perform on a resource. Actions can be common across tools (e.g., opening a file) but may also be specific to a tool or resource (e.g., committing files with `git`).

Friction. Extraneous actions and mental effort that arise when developers' goals cut across applications and resources in the environment.

3 Related Work

Prior work has investigated developers' tasks from different perspectives including low-level tool interactions, higher level development activities, and the way developers structure their goals. Additional studies have examined the challenges developers face during their tasks and have created approaches to address some of these challenges.

3.1 Tracking IDE Interactions

IDEs bring together many of the different tools and resources developers require and have been extensively studied. For example, Murphy et al. instrumented the Eclipse IDE to capture low-level interaction data to understand which IDE features developers use (Murphy et al, 2006). Similarly, Minelli et al (2015a) and Amann et al (2016) instrumented the Pharo and Visual Studio IDEs, respectively, to understand how developers spend their time, finding that as much as 40% is spent outside the IDE. It is thus important to investigate what developers are doing both within the IDE and externally to it to gain a complete understanding of how developers satisfy their goals. Even within the IDE, developers experience challenges and can become disoriented due to its bento box design where each file and tool is displayed in discrete and separate rectangular areas (DeLine and Rowan, 2010). This design leads to the *window plague* where many open tool and code windows make it difficult for developers to maintain an overview of their work and to locate and synthesize goal-relevant code and information (Roethlisberger et al, 2009; Minelli et al, 2015b). For example, Minelli et al (2015a) found that developers spent 14% of their time "fiddling with the UI" to organize windows within their IDEs. With the addition of eye tracking, Pilzer et al (2020) found that developers switched between windows every 16 seconds. Developers need to open and switch between numerous windows because a single code feature or concern is often spread across multiple artifacts each requiring its own dedicated window (Robillard and Murphy, 2007). This is also true of developers' goals, where information and actions are often spread across different tools and applications including the IDE, web browsers, and shells. Following prior observations that concerns cross-cut code resources, we investigate how goals cross-cut tools and the friction it creates.

3.2 Development Activities

Several previous studies have examined how developers interact with their tools to identify and understand higher-level development activities without having to consider their current context or goal. Researchers have devised many different sets of activities aligned to their data and research questions. For example, within the IDE, Minelli et al (2015a) categorized developer actions into four activities including navigation, understanding, UI interaction, and editing, while Amann et al (2016) also considered project management and building. Bao et al (2018) created a tracking system to capture the actions developers perform both in the IDE and across their other applications and trained a classifier to automatically segment low-level activity data into six higher-level development activities including coding, debugging, and navigating. Singer et al (1997) instead identified fourteen activities by shadowing a developer over several months. They found that certain activities were not well supported by existing tools by examining company-wide tool usage logs. LaToza et al (2006) surveyed and interviewed developers at Microsoft to understand the relationships between nine predefined development activities, the tools developers use to complete these activities, and the mental models developers maintain as they switch between these activities. In contrast to these prior studies which aggregate developer actions into activity categories to report what developers do, we directly observe sequences of individual actions to understand developers' actual development goals and to identify challenges they experience while trying to accomplish these goals.

3.3 Developer Goal Structure

While examining the activities developers undertake is useful for categorizing how developers spend their time, they provide little insight into how developers structure their work goals; specifically, why developers perform their activities and how they fit into their work. Early on, Bannon et al (1983) examined the structure of users' shell actions and observed that users have clear, but interleaved, goals and that environments do not provide support to effectively manage the corresponding interleaved commands. González et al (2004) identified similar behaviour after observing developers and other knowledge workers organizing their physical and digital items and communications around their goals. Chattopadhyay et al (2019) identified six patterns developers use to coordinate their goals based on observations of ten developers working on self-selected implementation, refactoring, and debugging tasks. However, these prior efforts do not investigate the low-level actions developers undertake to complete their goals nor the challenges they encounter in doing so. Our work addresses this gap by investigating how developers' decomposition of their tasks into goals is affected by their choice of tool actions.

3.4 Information in the Development Environment

Information foraging theory (IFT) is a theory that seeks to explain how people seek information (Pirolli and Card, 1995). In IFT, developers forage for information in *patches* which, in this study, correspond to the resources displayed in application windows. Developers can either locate their information *within* a window, move *between* windows, or *enrich* the information shown within a window by performing actions. For example, a developer wishing to reproduce a bug would first read the bug report (within-patch) and then switch to the shell (between-patch) to build and run the program (enrichment). If the program is a web application, the URL, shown in the run output, acts as a *cue* which prompts the developer to follow the *link* to the running application. The developer would then switch between the bug report and the application to follow the steps-to-reproduce (enrichment) until they observe the bug in the window (within-patch).

Prior work examining IFT, have limited the patches to individual applications (e.g., the IDE; Piorkowski et al (2013, 2015)) or specific types of resources including source code files (Lawrance et al, 2007; Piorkowski et al, 2016) and version control logs (Ragavan et al, 2021). Lawrance et al (2007) found that the words used in bug reports can be used to predict which classes developers are likely to visit. Within the IDE, Piorkowski et al (2016) found that misleading cues, information being scattered among patches, and the disjoint topologies used by different patches caused high navigation costs. We extend this work by investing how these challenges manifest and impact developers in the novel context of their cross-application environments. We discuss our findings in relation to these prior findings in Section 8.

3.5 Tool Challenges

Prior research has also noted difficulties developers experience translating and completing their goals using their tools. Maalej (2009) identified eight integration problems developers encounter when employing the average four to five heterogeneous tools needed to link information between their resources. Sillito et al (2008) noted that developers often have to break down their questions about program structure into tool-oriented questions and then attempt to reintegrate the individual, potentially incompatible, answers. Similarly, Ko et al (2007) identified the information developers need to avoid being blocked on their programming tasks and noted that, to access this information, developers have to “translate their questions into an awkward series of actions” across multiple resources.

For these scenarios, researchers have developed specialized tools and shown that they allow developers to complete their goals easier; for example, by allowing developers to locate information across siloed resources (e.g., Čubranić and Murphy (2003); Venolia (2005); Begel et al (2010)), organize and integrate resource fragments (e.g., DeLine and Rowan (2010); Bragdon et al (2010); Henley and Fleming (2014); Adeli et al (2020); Fritz and Murphy (2010)), and

transfer content between resources (e.g., Chapuis and Roussel (2007); Zhao et al (2012)). Other researchers have created approaches to allow developers to associate their resources with specific tasks to facilitate task switching and organization (e.g., Kaptelinin (2003); Kersten and Murphy (2005); Dragunov et al (2005); Smith et al (2003); Jeuris et al (2014); Robertson et al (2004); Tashman (2006); Rattenbury and Canny (2007); Bernstein et al (2008); Oliver et al (2006, 2008)) based on the concept of activity-centric computing (Bardram et al, 2019).

However, these studies focus on the information developers need and do not directly investigate the mechanics of how developers obtain the required information from their tools. We look specifically at these cross-application and cross-tool challenges, systematically identifying seven tool-centric frictions which add complexity to developers' workflows and reduce the speed at which they can complete their goals. These observations inform three design recommendations meant to specifically address frictions arising when developers use multiple applications for their goals as we describe in Section 7.

4 Methodology

To learn how developers decompose and operationalize their tasks into goals and actions, we adopted a mixed-methods approach using data collected from both fixed and user-selected tasks. Our approach was inspired by the Cognitive Walkthrough method where an analyst selects a specific task, determines an ideal sequence of actions, and identifies and reasons about trouble spots by tracing the mental process of a user (Lewis and Wharton, 1997). However, we apply these steps to actual users rather than working through the tasks ourselves. Specifically, we conducted a controlled user study to gain an understanding of how developers approached a fixed task and the challenges they encountered in doing so. To account for the effects of the controlled task and environment on developers' behaviour, we also collected and examined live-streamed recordings of real developers working on their own tasks in their own environments. A high-level overview of our mixed-methods approach is captured in Figure 1.

4.1 Controlled User Study

Our first study sought to gain insight into how developers decompose and operationalize a known fixed task (Figure 1, Step 1). We used a fixed task for all participants to remove task-induced confounds so we could confidently infer the goals and challenges of participants while allowing us to compare tool usage strategies between participants. Controlling the development task also allowed us to create a baseline workflow against which we could compare participants' approaches to convey a sense of the different ways developers breakdown their tasks and the work involved in completing them.

4.1.1 Participants

We recruited a total of 17 software developers (3 female, 14 male) through personal contacts and recruiting emails. Ten of our participants were professional software developers while the remaining seven were computer science students (one undergraduate, six graduate) experienced in software development. On average, participants had 15.1 ± 10.0 years of programming experience and 8.2 ± 7.3 years of professional experience. We compensated each participant with a $20 Amazon gift card for their time.

4.1.2 Project and Task

To gain insight into how developers work, we selected a bug fixing task from a real software system that we believed participants would be able to fix within 30 minutes. The bug[1] was reported on *Kanboard*, a medium-sized (230 KLOC) PHP project, which we identified by manually inspecting the recently opened issues of GitHub repositories labeled with the `beginner` topic. We selected this bug because the author included a suggested fix that was simple to understand and test while still complex enough that participants had to use their tools to complete it.

[1] https://github.com/kanboard/kanboard/issues/4213

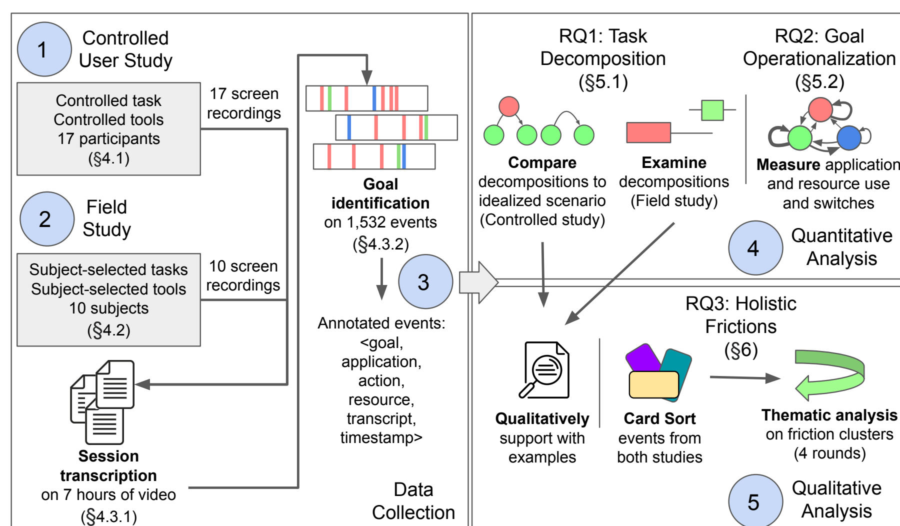

Fig. 1 The mixed methods research approach used. The order in which the steps were performed are denoted ①–⑤.

4.1.3 Idealized Workflow

The controlled nature of the task enabled us to create an idealized workflow consisting of the minimum number of actions developers would have to perform to complete the task. Table 1 provides a high-level overview of the goals a developer could follow to fix the defect. The table also includes the number of resources (i.e., files, shell commands, and web pages) they would use to perform each goal.

Table 1 An idealized workflow to complete `Kanboard` issue #4213. Numbers on the left of ▷ indicate the average number of resources used by participants in the user study. Numbers on the right indicate the ideal number of resources assuming no missteps, errors, or distractions.

		Application (Avg ▷ Ideal)		
Goal	**Description (Main Resources)**	**Browser**	**Shell**	**IDE**
G1	Learn what bug is about (Bug report)	4 ▷ 3	0 ▷ 0	0 ▷ 0
G2	Start test environment (README, `docker`)	5 ▷ 0	5 ▷ 3	1 ▷ 0
G3	Reproduce defect (Kanboard program)	16 ▷ 5	0 ▷ 0	0 ▷ 0
G4	Find defect in source code (Program files)	4 ▷ 0	2 ▷ 0	6 ▷ 2
G5	Review fix suggestion (Bug report)	1 ▷ 1	0 ▷ 0	0 ▷ 0
G6	Fix bug (Program file)	1 ▷ 0	0 ▷ 0	2 ▷ 1
G7	Check fix works as expected (`docker`, Program)	4 ▷ 1	3 ▷ 2	1 ▷ 0
G8	Share fix as a pull request (`git`, PR form)	4 ▷ 1	10 ▷ 4	0 ▷ 0
	⟨participant average⟩▷⟨total ideal⟩	40 ▷ 11	20 ▷ 9	10 ▷ 3

During goals G1–G3, the developer learns about the defect and reproduces it. This allows them to confirm that the problem actually exists, they are able to observe the problem, and they have a procedure they can use to verify that their changes successfully fix the problem. Ideally, this would involve looking at three different web pages (the issue report, project README, and project documentation) in the browser and switching to the shell and executing three different commands (learned from the web pages) to build and start Kanboard using its containerized Docker-based test environment. Finally, they would launch Kanboard in the browser to manually verify that they could reproduce the defect.

Goals G4–G6 involve actually fixing the fault; it is here where the idealized workflow hides most of the task's complexity as the developer would need to find the affected file on their first try and fix it perfectly. The developer searches in their IDE and finds the file causing the defect. They revisit the bug report to review the suggested fix and then switch back to the IDE to implement it.

During goal G7, the developer confirms that the fix worked. This involves rebuilding the `docker` image in the shell, relaunching the test environment, and switching back to the browser to verify that the fix was successful.

Goal G8 involves submitting the changes for the fix. First the developer performs four steps in the shell: checkout a branch, stage, commit, and push

the changes to the remote server. Then they follow a link to GitHub's web interface to create a pull request for this change in the browser.

Fixing this simple fault needed changes to only two lines of code, but the developer had to decompose the task into eight goals which they would operationalize by switching between three applications nine times to access twenty unique resources a total of twenty-three times. The idealized workflow assumes the developer switched directly to the required resources and remembered all of the required information as they switched between applications and goals.

4.1.4 Method

We copied the Kanboard project to a new GitHub repository created specifically for this study to avoid affecting the real project and to eliminate any anxiety participants might experience sharing their changes. We piloted the task with an experienced developer, who did not take part in the study, to verify the task difficulty and duration. Based on their feedback, we added a steps-to-reproduce comment to the issue description on GitHub with screenshots of the steps necessary to see the bug in the Kanboard user interface. This reduced the overall time participants needed to spend on the task while still allowing us to observe participants using all of the tools necessary to complete the task.

We met with each participant individually at a location of their choosing. The first author was present during the study to take notes and answer any questions that arose. Each participant was provided a MacBook Air configured to record the screen and track interactions with default installations of IntelliJ, Visual Studio Code, Google Chrome, Firefox, and Terminal. The trackers recorded the URL and title for web pages participants opened in the browser, the path of files opened in the IDE, and the session, working directory, exit code, and command line for commands executed in the shell. Participants were free to configure the environment as they wished but we requested that they use only the aforementioned applications to complete the task.

Before starting the study, we briefly described the project and showed participants a task board containing a link to the issue they should work on. We told them that they should try to have the task done in 30 minutes and that the task was considered complete once they had opened a pull request with any changes they felt solved the issue. We also explained that the project had already been cloned and configured locally and that they would be working in a sandboxed environment. To help participants start the task and get familiar with the project, we requested that they begin by reproducing the issue. If a participant got stuck or sidetracked we provided guidance to ensure they were able to complete the entire task so we could observe the full diversity of their tool interactions.

Table 2 Presenters' tasks and development experience in years.

Sub	Exp	Role	Time	Task Description
S1	7	Senior Developer	14m30s	Alter table sorting behaviour.
S2	10	Senior Developer	9m28s	Stop streams when window is closed.
S3	25	Technical Trainer	27m45s	Create a new Java Spring project.
S4	15	Cloud Advocate	15m10s	Understand Twitch service response.
S5	16	Senior Developer	7m55s	Release new version of Azure plugin.
S6	10	Developer Evangelist	26m49s	Create serverless blog search feature.
S7	15	Outreach Manager	17m00s	Migrate ASP.NET button to Blazor.
S8	15	Developer Advocate	6m32s	Refactor and deploy REST endpoint.
S9	6	Developer Evangelist	19m02s	Obfuscate tokens in log output.
S10	1	Developer Evangelist	16m26s	Create README and LICENSE.

4.2 Field Study

We use live-streamed screen recordings of ten developers (Figure 1, Step 2) to augment the recordings made during the controlled user study and strengthen the ecological validity of our observations. Researchers have found these recordings are not rehearsed and illustrate developers' real contributions to software projects using their preferred environment (Alaboudi and LaToza, 2019). Exchanges between the developers and their audience provide running commentary, similar to think aloud, describing what they are doing and why.

4.2.1 Subjects

The subjects[2] for this study were ten developers and a section of one of their recently streamed development sessions. Table 2 provides information about the presenters and their selected sections.

We identified developers and videos through an iterative search and review process. Videos had to be presented in English in a professional manner with a clear software development task which the presenter attempted to solve during the video. We excluded tutorials, Q&A and office-hour-type sessions, and videos designed specifically to demonstrate a feature or procedure. We started by examining the most recently published videos from members of the Live Coders team as they are required present in a professional manner, write code regularly, and to have had at least five viewers in the 30 days prior to the review of their application.[3] Six of the developers were members of the Live Coders team. We identified the remaining four developers by searching for *live coding* videos on Twitch and YouTube. Our final set of videos capture a diverse set of tasks, environments, and program languages, sizes, and maturities.

Videos ranged from 1.5 to over 3 hours in length. During this time presenters worked on multiple tasks but always clearly described the task on which they were currently working, either orally or textually with issues or TODO

[2] We label the people evaluated in the controlled study *participants* (P1..P17) and those from the field study *subjects* (S1..S10) throughout the paper.

[3] `https://livecoders.dev`

items. We scanned through the videos identifying sections in which the presenter switched between multiple applications so that we could extract the maximum amount of information about the cross-application friction developers experience across a range of tools. Due to the frequency of these sections and the amount of work required to create a detailed transcript of the presenters' numerous low-level actions, we selected only one section for each presenter. We ensured the selected section encompassed a complete task to help us infer the presenter's goals.

Before finalizing our video selection we verified that the presenters had professional software development experience using the information provided on their LinkedIn profiles. The ten presenters (9 male, 1 female) each had at least one year of development experience and an average of 12 ± 7 years.

4.2.2 Projects and Tasks

We observed subjects working on both personal and open source community projects. S1, S4, and S8 worked on bots that respond to commands issued by viewers through the Twitch API. S3 created a new Java project to compute stock options, S6 implemented a website search feature using Azure's Cognitive Search API, and S10 worked on a pull request tracker. S2 worked on the Mozilla Firefox browser project, S5 worked on C# Make integration for Azure DevOps, S7 worked on a project to recreate WebForms components in Blazor, and S9 worked on Twilio's CLI for deploying serverless functions. The specific tasks each subject worked on during the selected section of video are shown in Table 2.

4.3 Data Preparation

Data from the two studies was processed through three high-level steps to prepare it for analysis. The anonymous transcribed data is available online for further analysis.[4]

4.3.1 Transcribing sessions.

In total, we collected more than 7 hours of screen recordings across the 27 subjects from both studies with an average duration of 16 ± 6 minutes. The first author transcribed events from the videos by recording the timestamp, resource identifier, application, and action performed each time a subject switched resources. In total, 1,532 events were transcribed from the screen recordings.

Separately from the transcripts, the first author also identified and recorded instances in which developers performed unexpected actions or encountered challenges. These included instances in which developers repeated actions or sequences of actions, switched quickly between windows, switched back and

[4] `https://osf.io/68qxg/?view_only=435d824fe8ee493089e66a7cc7fc5b08`

forth between resources, performed actions that failed (either explicitly with an error message or implicitly by not performing the desired operation), performed more actions than necessary to accomplish a goal, or when developers stated that they or their tools made a mistake or did not work as expected. It is from these observations that we derive developer friction (RQ2).

4.3.2 Identifying goals.

After transcribing the screen recordings, our dataset consisted of lists of actions performed by each developer. To understand how developers decompose their tasks into goals and how directly they can complete these goals in their environments (RQ1) we had to segment the transcripts into goals (Figure 1, Step 3).

When deciding where to segment goal boundaries in the transcripts, we considered each action in the context of its neighbours and how it was used by the developer. We examined the transcripts both forwards, following the actions the developers performed inferring what they were trying to achieve, and backwards, working from the results and inferring which steps were used to achieve them. We referred to the screen recordings throughout the process to ensure that we correctly interpreted the actions performed by the developers. We also looked for and used cues indicating developers' intentions when they were present. These included subject utterances and think aloud, e.g., "What we need to do now is implement actual searching." (S6), text they highlighted or copied, text they made visible by scrolling, keywords and search terms, and the resources they opened and closed.

Two authors independently segmented three action transcripts from the field study establishing both goal boundaries and descriptions. In 16 out of 28 cases, the coders fully agreed and in 4 cases they were off by one action. In 8 cases one of the coders segmented the actions using an additional goal. All three authors discussed which aspects of the actions and cues resulted in disagreements among the eight goals. Through this process of negotiated agreement (Garrison et al, 2006), we were able to agree on how we should interpret actions in relation to the goals. However, identifying the precise location of goal boundaries was still a subjective process. To ensure consistency between the goals, the first author segmented the remaining transcripts following the agreed upon interpretation.

5 Making Tasks Actionable

Ultimately developers want to complete their tasks. But tasks can rarely be done with one action. Instead, developers decompose their tasks into a series of manageable goals. Since each of these goals still cannot be directly executed by traditional environments, developers further operationalize each goal: they manually decompose their goal into a series of actions that they can perform using their environment. These two levels of decomposition (from

task-to-goal and goal-to-actions) are not linear sequences: developers often encounter problems which can divert them as they solve new goals or devise new operationalizations. The decomposition process also requires explicit developer effort as they reason from their task through goals to the low-level actions they perform to complete their work. The workflows of the ten field study subjects are shown in Figure 2. We describe their task decompositions and goal operationalizations in the following sections.

5.1 Decomposing Tasks to Goals

Developers decompose their tasks into goals to provide intermediate objectives between the high-level task description and the low-level actions they need to perform to complete the task. This aligns with research on the cognitive process of problem solving in software development (Wang and Chiew, 2010) and how people conceptualize their actions (Vallacher and Wegner, 2012). In this work we consider goals at the level of abstraction that developers would use to communicate with each other. We found that task decomposition happens on-demand and results in similar sequences of goals for developers completing the same task (Figure 1, Step 4).

5.1.1 Developers decompose the same task into similar sequences of goals

The controlled nature of the user study provided an opportunity to examine how participants decomposed their relatively straight-forward task into goals. We found that the sequence of these goals was generally similar among participants and the idealized workflow described in Section 4.1.3. Specifically, all but three of the participants ordered their goals in the same way: P8 interleaved goals while waiting for a command to finish, P10 choose to understand the proposed fix before reproducing the bug, and P17 choose to start the test environment before reading the issue.

However, not all participants used the same number of goals to complete the task. Nine of the participants decomposed the task into one *fewer* goals by using keywords from the suggested fix to locate the bug directly without a dedicated defect reproduction goal. Some participants also underwent *additional* goals when they sought a holistic understanding of the project before attempting to start the test environment (P1, P6, P7, P10, P11), and when they had to re-attempt failed goals, for instance after implementing the fix in the wrong code location (P13, P16) or attempting to reproduce the bug before starting the test environment (P12).

5.1.2 Developers formulate goals on-demand

Occasionally, developers encounter unexpected obstacles that they must overcome to continue making progress on their original goal. In these cases, they

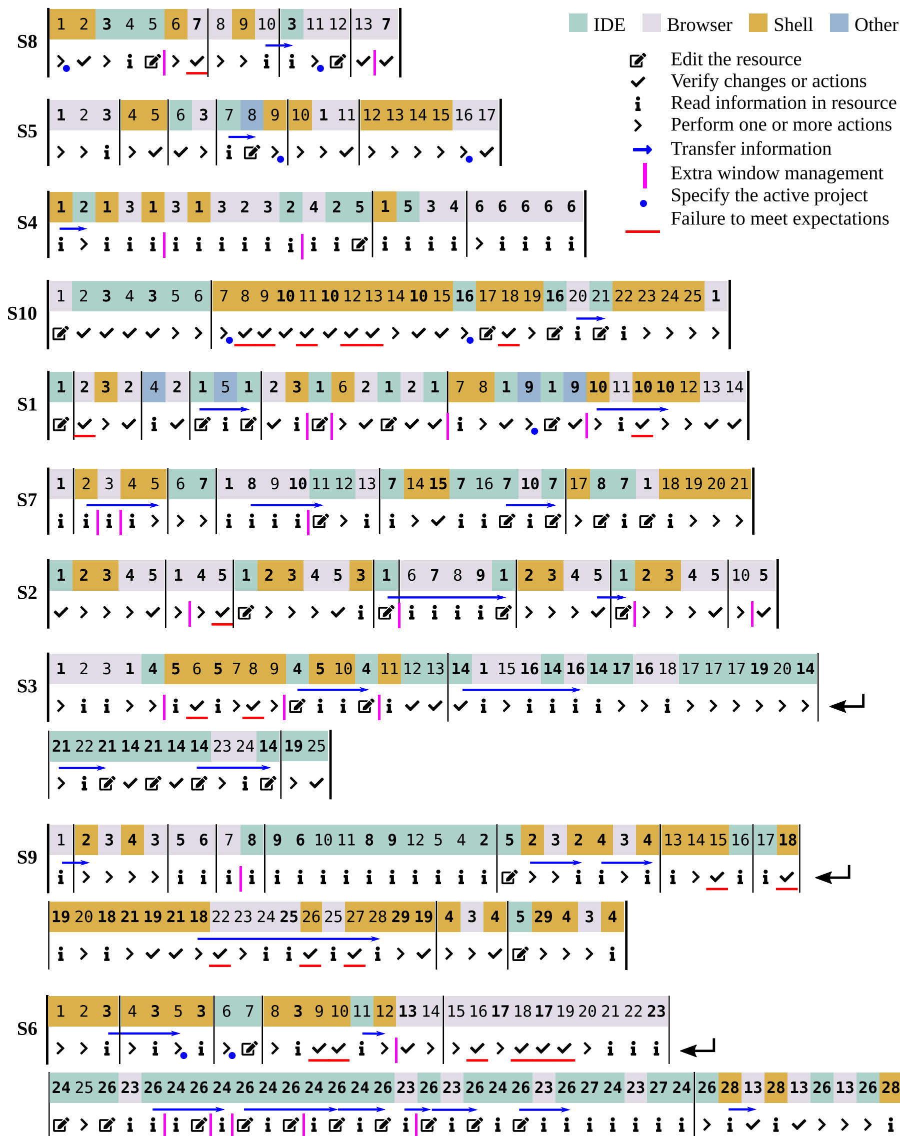

Fig. 2 Workflows used by developers in the field study to complete their tasks. Vertical bars indicate goal boundaries. Numbers identify individual resources and the coloured background indicates the associated application. Revisited resources are shown in bold.

formulate new goals to overcome these obstacles. For example, when attempting to create a branch using `git`, P4 encountered a error stating she had provided incorrect flags. She responded that she knew "there was a way to do it" and proceeded to consult the help page, trying several variations of the command, before she was successful and could resume her original goal of sharing her changes.

There were also many instances of developers formulating on-demand goals in the field study. While debugging, S2 remarked "I didn't hit my breakpoint" and formulated a new goal to investigate whether the debugger "hits the unload event if you close a window." S3 formulated a new goal to investigate "why that [package version] is red? That shouldn't be red" when he noticed that the POM file in his Java project reported and error. He had to complete this emergent goal before he could finish creating his project and complete his original goal.

5.2 Operationalizing Goals to Actions

After decomposing tasks into goals, developers map these goals into actions that they use to interact with their resources. However, coming up with an effective mapping for a goal is not automatic: developers must consider what actions are available in each of their applications and how those actions can be combined across applications and resources to successfully operationalize their goal. Developers also have to handle discrepancies between their expectations and the actual result of their actions. We describe the factors affecting developers ability to map and operationalize their goals below (Figure 1, Step 4).

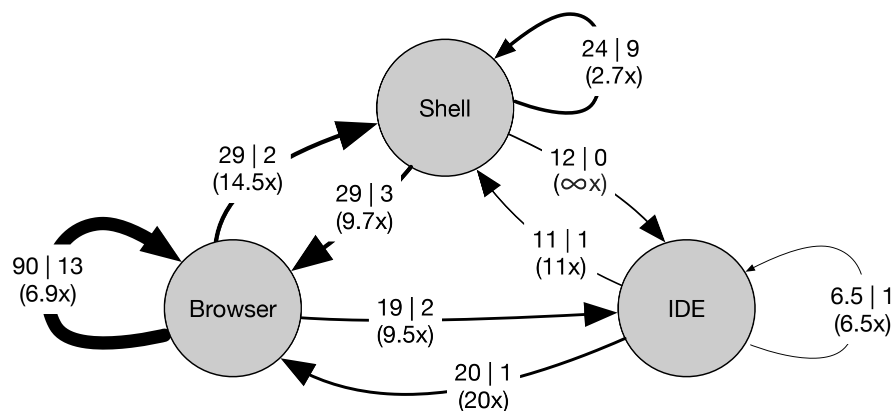

Fig. 3 Resource switches. Numbers include the average number of times each kind of switch happened, the idealized number of times a switch would be needed, and the proportion of switches in excess of ideal. Self-loops indicate that subjects switched between resources within the application (e.g., from one web page to another), while arrows between nodes indicate that subjects switched from a resource in one application to a resource in another application.

5.2.1 Environments do not allow developers to directly express their higher-level goals

Instead of directly communicating their goals to the environment, developers have to mentally map their goals to the low-level actions offered by the environment. This process is not trivial as the low-level nature of actions means developers have to consider and use many actions for each of their goals. On average, participants used 8.1 ± 6.5 actions per goal, using a maximum of 36 actions for a single goal. The need to consider so many actions imposes additional overhead as developers need to know how the actions can be effectively composed. For example, the excess switches in Figure 3 were the result of participants using extra actions to overcome incomplete mappings of their goals. This overhead is also evidenced by the actions that did not work as participants expected (shown using a _ in Figure 2).

5.2.2 Developers' knowledge and preferences of application differences affect their operationalizations

The applications developers use each have different user interfaces and set of actions. While some applications share similar interfaces at a high-level (e.g., tab-based web browsers), default key-bindings, parameters and syntax of text-based shells and even the interfaces of individual web pages vary. These interfaces each have individual strengths and weakness which developers must consider when operationalizing their goal. For example, P16 believed that it would be easier to run the project's docker commands from within his IDE rather than switching to the shell so he decided to spend extra time configuring his IDE's docker component.

Multiple applications often overlap in their functionality, requiring developers to choose among alternatives. For example, two of the user study subjects (P8, P16) and 1 of 4 subjects (S10) in the field study that used version control during their task, used the IDE, while all other participants chose to invoke git directly in the shell. Similarly, some subjects searched through their code using the GitHub repository search (S9), shell-based grep (P2), and using the IDE (all other subjects). These choices impact the number of actions developers have to consider when operationalizing their goal. For example, when committing changes, participants who chose to use the shell had to perform actions to see their outstanding changes and to verify they had been committed while this information was provided automatically in the IDE interface.

5.2.3 Operationalizing goals typically requires multiple applications and resources

Developers primarily use three kinds of applications: web browsers, IDEs, and shells. They used these applications to work with resources including code files, documentation, log files, and command output. Table 1 shows how participants used these applications to complete the goals of the task. Overall, participants

in the user study used their browsers to view 677 (57%) resources, their shells to view 334 (28%) resources,[5] and their IDEs to view 174 (15%) resources, while subjects in the field study viewed their resources evenly across the three applications.

Beyond simple interactions with their environment to switch between windows, developers had to perform actions in multiple applications across multiple resources to complete their goals. Figure 2 shows the applications, resources, and actions used by the subjects in the field study. The alternating colouring indicates that developers do not use applications sequentially but rather they need to switch between applications to interact with different resources while operationalizing their goals.

5.2.4 Developers must continuously locate and manage resources across applications

Developers need to locate, examine, modify, and manage many resources to accomplish their tasks. Managing resources is challenging because developers have to locate them before they can be opened, decide how they should be organized, and determine when they are no longer relevant and should be closed. This overhead increases with the number of resources used in a goal as the environment becomes cluttered. Subjects in the user study used an average of 25.7 resources to complete the task and 3.4 resources to complete each goal. Subjects in the field study used an average of 18.3 resources to complete their tasks and 2.7 resources to complete each goal.

To access their resources, developers have to frequently switch between different application windows and tabs (Figure 3). Over the course of completing their tasks, subjects switched between their resources an average 70 times (9 times per goal) in the user study and 34 times (6 times per goal) in the field study. Developers often switch to revisit resources, either to recall previously used information or to obtain new information from other areas of the resource. For example, subjects revisited the issue description to recall the suggested fix and to copy the exact keywords required to locate the source of the bug. Revisits are depicted in Figure 2 by boldface numbers. On average, subjects revisited resources 3 times per goal in the user study and 2 times per goal in the field study.

In some cases, only a small portion of a given resource is relevant to a goal. For example, subjects in the user study only used 2 of the 45 lines in the patched source code file, and only needed one sentence and 2 command descriptions from the whole project documentation. Developers have to spend time and mental effort getting to these pieces of information by first identifying the resource containing the information using a potentially unhelpful descriptor (e.g., a title or file name) and then reading, scrolling, or searching the resource to find the relevant content for their goal.

[5] Including the IDE-integrated shell.

5.2.5 Information flows between applications and resources

The information developers need is spread across multiple resources and applications. This means that developers have to manually seek out and move the information between resources. They do so by using copy and paste or recalling the information from memory (sometimes with the help of the environment; e.g., auto-completing directory paths).

Figure 2 shows instances where it was apparent that developers sought out and moved information between resources verbatim. Developers moved information under two scenarios. In the first scenario, developers used information they observed in a previously viewed resource in a later resource. This is the case when the arrow starts and ends on different resources. For example, S6 used information (directory names) from the output of `ls` to recall and change into his project's directory (depicted by the arrow from resource 3 to resource 5). In the second scenario, developers realized they needed some information in a particular resource and had to perform actions to obtain the information before moving it into the original resource. This is the case when the arrow starts and ends on the same resource. For example, when configuring his project to use the latest version of Java, S2 had to leave his IDE to find the installation path and replicate that path in the configuration dialog.

Developers also move information implicitly across resources. For example, S4 used the property name of an object to inspect debug output, understand the structure of the object by searching for the name in online documentation, and to follow a hyperlink describing the property in detail. We do not show these implicit movements in Figure 2 since they require interpretation of the information and where it originated (i.e., the information may be abstract and not directly visible in the resource). It is not clear, for example, how S4 knew which property to investigate based on the resources he previously used.

5.2.6 Discrepancies between expected and actual results of an action cause additional actions and adjustments

The actions developers perform do not always work as intended. Figure 2 shows the actions that failed during the field study (using a _).

The most common failure encountered by developers was due to invalid shell commands (48%) (S6, S9, S10). Developers also encountered failures identifying the resources that contained the information they needed (26%) (S3, S6, S9) and when their actions behaved differently than expected (17%) (S1, S2, S3, S9). For example, S1 accidentally aborted a commit when he used the wrong key sequence to exit `vi` while S3 had to guess an alternative parameter value when a previous attempt failed. Finally, there were two instances where developers attempted to verify a code change by observing the output but failed to ensure the necessary preconditions were met (9%); e.g., that the development server was running (S1) and that the required environment variables were set (S8).

Sometimes it is not apparent when an action did not work as expected. In these cases, developers have to explicitly verify the effect of their actions, which are shown as a ✔ in Figure 2. Sometimes verification requires only a single action (e.g., running `git status` to ensure the commit action committed the correct files), but other times it can be a time consuming process (e.g., when manually testing a program change). Of particular interest are cases where developers have to re-verify their changes, repeatedly performing the same sequence of actions. For example, S2 modified his code five times during the development session and each time had to perform nine actions across both the shell and browser to verify the change.

RQ1 Summary

Developers use similar sequences of goals when decomposing the same task, formulating goals on-demand to overcome unexpected obstacles. Operationalizing goals to actions is an indirect and personal process complicated by the need to work across applications.

6 Friction

To complete a task, developers decompose the task into goals and then operationalize these goals by performing sequences of actions across applications in the environment. Since developers' goals frequently cut across applications and resources, performing these actions can be difficult and induce friction that manifests itself as extraneous actions and mental effort. In this section, we describe the frictions developers experience when interacting with their complete environments, an area which has been largely neglected in favour of more targeted studies of individual applications and tools.

To examine the friction that occurs due to the mismatch between developers' goals and the actions available, we analyzed the video recordings and transcripts from both studies using a grounded theory methodology Strauss and Corbin (1994). After identifying the goals of each developer, the first author recorded descriptions of the actions developers performed that did not directly contribute to their goal and organized these descriptions into instances of friction. All three authors then conducted four rounds of thematic analysis categorizing the instances of friction into three higher-level themes that describe the ways developers interact with their environments: translating, integrating, and accessing resources (Figure 1, Step 5).

We observed a total of 386 instances of friction (231 instances in the user study and 155 instances in the field study). A summary of the frictions and their distribution are provide in Table 3.

Table 3 A summary of frictions in cross-application workflows. *Cause* summarizes how developers encounter these frictions and *Workaround* exemplifies how developers manually overcome the friction.

		Burden	Cause	Workaround
Low-Level Actions	Translation Friction	Plan Workflows 59/386; 15%	Crucial information for completing goals is often not completely represented in a fixed resource (e.g., who last changed a file). Developers must devise workflows to get this information considering the available tools.	Developers complete their goals opportunistically either using tools that are readily available or those they are familiar with. They seek out and use feedback about their actions to update their planned workflows.
Low-Level Actions	Translation Friction	Learn and Adapt 55/386; 14%	Developers must learn how they can achieve a goal using the available actions. They form expectations about their tools which may lead to errors and confusion if behaviour is not consistent between tools.	Developers use trial-and-error to learn (and relearn) the steps necessary to complete their goals in different applications. When a tool does not meet their needs, they often install new tools or plug-ins, or perform extra work to adapt it.
Dispersed Resources	Integration Friction	Switch 94/386; 24%	Developers often use multiple resources per goal which they must switch between. Standard navigation tools require developers to implicitly maintain relationships between resources, differentiate within- and across-application switches, and do not provide sufficient cues for developers to switch accurately.	Developers use CTRL-tab and ALT-tab to switch between their windows and tabs or visual search by manually clicking through their open resources. This process requires repeated switches which are distracting. Switching may fail, forcing the developer to re-open the resource.
Dispersed Resources	Integration Friction	Organize 40/386; 10%	Operating system windows open at fixed sizes and positions, regardless of why the window was opened or how the content will be used relative to other resources.	Developers manually arrange tabs and windows to put related resources closer together so they can use information from multiple windows simultaneously.
Dispersed Resources	Integration Friction	Transfer 61/386; 16%	The information developers need for their tasks is spread across different applications and tabs. This information must be manually integrated in order for developers to accomplish their goals.	Developers preemptively copy information they will subsequently need, or later navigate to a resource containing the information they require. This is usually accomplished with copy-and-paste.
Independent Applications	Access Friction	Navigate 45/386; 12%	Goals require developers to manually access related resources across applications. Developers must traverse different organizational structures starting from application-dependent fixed locations (e.g., the shell's default directory or browser home page).	Developers manually navigate from the fixed default locations to their project location by either inputting the location directly or by using application-specific navigation steps (e.g., using `cd` in the shell or browser history).
Independent Applications	Access Friction	Configure 32/386; 8%	Applications need to be in a specific state for developers to perform actions and access information for their goal. This state is fragmented across applications, sometimes obscure, and changes as developers perform actions for subgoals.	Developers often emit details about their environment in logging statements to record and surface their values. They perform extra steps to set and verify their current project across all of the applications they use.

6.1 Translating Goals to Actions

The information and actions provided by the environment do not always align with developers' goals, forcing them to *plan workflows* that translate their goals into low-level actions supported by the environment. This mismatch between the environment and developers' goals means developers often need to *learn* how the available actions can be made to satisfy their needs, or *adapt the environment* to better conform to their desired workflows.

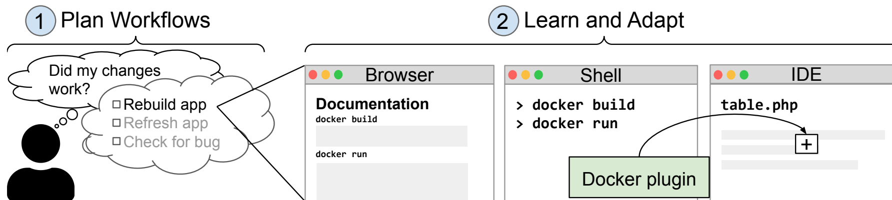

Fig. 4 Developers must translate their goals into actions. They need to ① mentally decompose their goals into a plan the actions. This can require developers to ② learn the exact details of the commands or to adapt their tools to simplify the actions.

6.1.1 Planning Goal Workflows

The information developers need to accomplish their goals is often not represented explicitly in the environment. Developers frequently need to devise workflows to answer high-level questions with the tools that are available to them. While some questions have answers that are available directly in the environment, answering them still requires effort. For example, questions like "What branches do I have [locally]?" (S7) and "Where is [Java] installed?" (S3) can be answered by single commands, but identifying and executing the commands can interrupt developers' tasks as they have to switch to separate applications and transfer the answers to where they are needed.

Unfortunately, many of these questions require more complex workflows for which developers need to seek out and synthesize multiple resources. One such case arose in the user study when subjects needed to know if their patch successfully fixed the issue. Since the environment does not provide any way for developers to ascertain this information directly, subjects had to devise a workflow that included building their app in the shell, refreshing it in the browser, replicating the issue in the browser, and comparing the updated output with the screenshot in the issue tracker to actually answer their question (Figure 4, ①). When S3 got an error in his IDE he wondered "why is that [dependency] red?" He had to check online that he was using the correct version, and then had to refresh the dependencies, recompile the project, and invalidate the IDE caches. Ultimately, it was an error in his IDE but it took over seven minutes of attention to determine whether it was actually an error.

The information provided by environments can sometimes be ambiguous. When feedback is not as developers expect, they have to determine if they were expecting the wrong information or asking for the wrong information. For example, subjects in the controlled study expected that their code changes would be automatically shown in the Kanboard web application based on their experience with other web frameworks. Instead, Kanboard required the subjects to relaunch the development instance after every change. The uncertainty of this expectation caused participants to wonder whether the page had actually changed and resulted in them spending extra time inspecting the page's rendered HTML code and comparing it with the screenshot provided in the issue tracker. S2, from the field study, was expecting that the breakpoint he set would pause execution allowing him to inspect the state of his application. When this did not happen, he had to determine what caused the breakpoint to not work as expected.

6.1.2 Learning and Adapting the Environment

Developers form expectations about their environments that can lead to errors when they are incorrect. While errors help alert developers that their expectations are wrong, they do little to help developers effectively change their expectations. Instead, developers have to go through a tedious and time-consuming process of learning and correcting their expectations to conform to the environment's conventions. For example, S10 expected the `touch` command would create an empty file but instead she received an error that the command is not available on Windows. She had to use six actions to learn an alternative method for creating a file. Similarly, P9 and P11 used eight and eleven actions, respectively, to learn how to move their commit to a new branch. Even experienced developers working in their own environments form incorrect expectations which they have to correct. For example, when attempting to install the latest Java version, S3 stated he "always forgets the commands" which caused him to run five unnecessary commands despite reading the help page three times. S9 expected the action `npm link` would create a binary but it took twelve commands, two Google searches, and reading a Stack Overflow and blog post to determine that the correct action was the very similar `twilio:link`.

Instead of adapting to the environment's conventions, developers can change them through customization. This requires developers to be aware of their work habits and to seek out and incorporate the changes into their environments. For example, P16 digressed from his goal to spend more than 3 minutes configuring his IDE to run Docker directly so he would not have to work in the shell (Figure 4, ②). Similarly, P1 took time from his goal to alter the behaviour of his IDE by installing a VIM plugin while S8 configured his shell to use an advanced version of `cd`. Regardless of whether developers choose to learn existing conventions or to customize them, they are both active processes that take developers' time and attention away from their goals.

6.2 Integrating Resources

Developers have to piece together information from multiple resources, such as source code files, issues, and Q&A websites, using various applications to accomplish their goals. Integrating information from multiple resources and applications can induce friction, especially when developers *switch*, *organize*, and *transfer content* between resources, requiring additional mental and manual effort. Figure 5 depicts these burdens for the reproduce and find defect goals of the user study task (Table 1, G3 and G4).

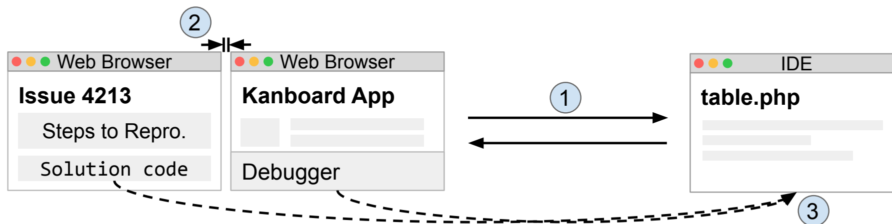

Fig. 5 Resources are dispersed in windows across the environment. To integrate these resources, developers manually ① switch between and ② organize windows to make information visible so they can ③ transfer it.

6.2.1 Switching Between Resources

The application and resource-centric nature of current environments commonly treats resources and applications as information silos offering only rudimentary support for the cross-cutting workflows used by developers. This *lack of cross-resource workflow support* often limits the simultaneous display of information and leads to developers frequently switching back and forth between resources to gather and integrate the relevant information located in various places. In both of our studies, developers frequently switched back and forth between resources. In the controlled user study for example, subjects switched an average of 13 times between the issue and the Kanboard program to apply the steps-to-reproduce, and S4 switched 10 times between an API response and its documentation to understand it.

This switching can incur a high cost, especially since it requires developers to get to the relevant resources again, often having to remember where they are and frequently resulting in “mis-switches”—switches to the wrong resource—that can further lead to distractions. Almost all developers in both studies had several mis-switch sequences (depicted in Figure 2 by |). Additionally, these ‘switch cycles’ often involve more than two resources, incurring an even higher switching and relocation cost. For example, S8 had to switch back and forth between six resources across three different applications to diagnose a bug. Another common switch cycle with multiple resources occurred when developers

engaged in workflows to gather feedback for their changes. For example, after S2 made code changes in the IDE, he switched to the corresponding shell to build the project, then to the web browser to verify the changes. He repeated this loop five times to complete his goal.

6.2.2 Organizing Resources

Tasks and goals often require accessing resources in multiple applications and at the same time, resources often crosscut goals. This crosscutting nature of tasks, goals and resources can induce friction by requiring developers to manually organize their resources. In our studies, participants organized their resources by re-ordering application tabs (P4, S6), moving windows to virtual desktops (P1, P4, P5, P6, P17), positioning windows to make the contents visible simultaneously (P1, P6, P9, P11, P14, S4), or even using an additional browser application as a makeshift way to separate and organize the project's online documentation from the web app they were debugging (P13, P16). In some cases however, even the organizational mechanisms were insufficient, resulting in a high cost for relocating a resource. For instance, S6 had created a project but could not recall how it was organized within the filesystem: "I know I created a [project]...so where did I store it?" He had to navigate between five different directories to locate his project which took over a minute and caused him some confusion "I feel like I'm missing something."

6.2.3 Transferring Content Between Resources

In addition to switching between resources, developers frequently leverage and reuse information between resources (depicted in Figure 2 by →). This information transfer induces friction as developers need to either remember the information and apply it to the other resource, or relocate the information manually using copy-and-paste, often also disrupting their flow of work. These situations occur frequently in any task, for example when developers copy and paste commands to run in the shell from some kind of documentation (e.g. S9), copy license information from files they worked on earlier (S10), or copy the branch name to use as a parameter in a shell git command (S7).

Transfer friction further increases when developers repeatedly transfer information and have to continuously relocate themselves or are transferring between resources that require complex switches. For example, S6 duplicated the structure of an existing source code file in a new code file by repeatedly switching back and forth between the two files, memorizing parts of the structure and recreating it in the new file. For each transfer the developer had to re-locate herself in the new file and recall what she was doing. Another common case across study participants was the transfer of the issue number into the commit message or a pull request description to link them. This transfer was performed by 14 subjects in the controlled and field study and required them to interrupt their current goal of committing code changes to switch to the issue tracker in the browser, find the issue, copy its number, and paste it

in the shell. To work around the current limitations for transferring information and to avoid disruption to their workflows, we observed some developers preemptively copying information. For example, P7 preemptively copied credentials listed in the documentation although they were never needed and might have overwritten other relevant information in the clipboard.

6.3 Accessing Information

One of the common themes of developers' actions is to get to a resource to extract information or make changes. However, the environment's siloed applications create friction making this information difficult to access. Even if developers know exactly which resource they need to accomplish their goals, getting to the resource in the environment is often not straightforward requiring them to *navigate to the resource* across different organizational structures. Developers also have to perform actions to *identify and configure application state* as a prerequisite to performing other actions. Figure 6 depicts the burdens developers experience accessing the information needed to complete their goals across the siloed applications in the environment.

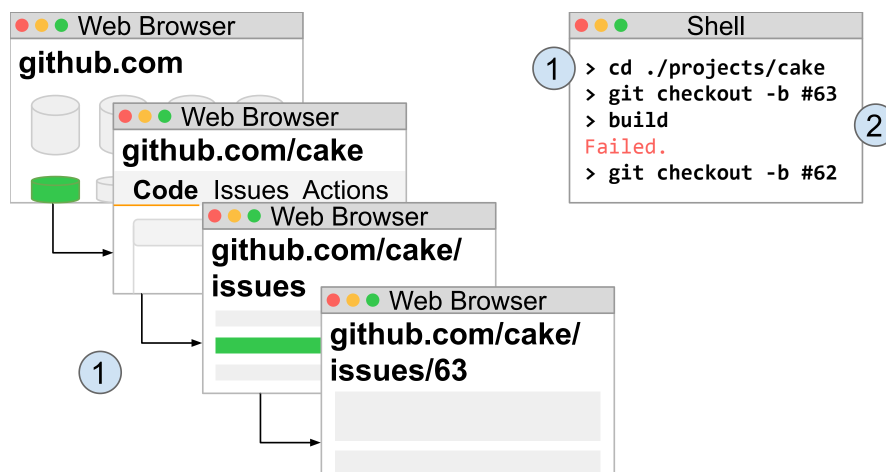

Fig. 6 Accessing information requires developers navigate different organizational structures while maintaining appropriate state. Obtaining different kinds of information such as the issue number or build status requires developers to ① navigate different organizational structures. State, such as the active branch, can ② affect the behaviour of subsequent actions in unexpected ways.

6.3.1 Navigating to Resources

Resources are stored and accessed in independent structures that vary by resource type, application, and location (remote or local). For instance, to get

the steps-to-reproduce contained in a task issue, the developer might open a new tab in a web browser window, go to her GitHub account, choose the 'Issues' tab and then the 'Assigned' tab within, scroll to the relevant issue, open it in the same tab and then browse the issue to find the reproduction steps, thereby navigating the structure of the tabs of a web browser and various (nested) structures within the GitHub service.

The *lack of uniform structures and mechanisms to access resources* requires developers to keep track of the different structures and mechanisms and correctly piece them together to navigate to the relevant resource. Developers frequently need to re-specify their project in different applications before they can access these resources. In our controlled user study, all 17 subjects navigated to the Kanboard project in both their IDE and on GitHub to access specific resources, and all but three also navigated to the Kanboard project in their shell, using an average of four `cd` and `ls` commands. Subjects in the field study had to navigate within even more applications, including Sublime Merge (S1), AppVeyor (S5), and AWS (S8) to access the relevant resources (depicted in Figure 2 by •).

Occasionally, developers go down the wrong path and have to go back or even switch applications. For example, after navigating to his project in his IDE, S9 starting looking for a necessary file by traversing the project's directories. He quickly gave up on this strategy and instead switched to GitHub where he navigated to the 'Code' tab. Using GitHub's code navigation feature, he traced through a series of method definitions to locate his desired file. He then had to manually navigate to the same file in his IDE so he could make changes. Just to locate a relevant file, S9 had to navigate different organizational structures in two separate applications using different navigational mechanisms.

Given the high number of resources and the high frequency with which developers access resources to complete their tasks, the induced friction can incur a high cost.

6.3.2 Identifying and Configuring Application State

Applications need to be in a specific state for developers to run actions successfully. For example, the shell's working directory determines where an action is run. However, the state of an application is not always apparent and remembering the state accurately can be difficult for developers. This is especially true when switching between applications and resources since they act as silos which fragment the state needed to complete a goal. As developers perform actions, they also inadvertently alter the environment's state making it difficult to manage.

When the actual state of an application is different from what developers assume they encounter unexpected behaviour from their actions. This means developers either have to explicitly seek out the current state of their application or handle the behaviour of any actions they perform. For example, when the working directory of the shell was set outside of his project, P2 had to

investigate the large number of irrelevant search results returned by `grep`. S8 had to trace through his program to identify the name of an environment variable he suspected to have caused an error. He then had to manually navigate through a web portal to locate and set the correct value for the variable. Had the environment variables been visible, this could have been resolved directly.

RQ2 Summary

Environments often force developers to perform extra work to accomplish their goals. These extra steps introduce friction into developer workflows by requiring developers to adapt to their environments (translation friction), integrate information between resources (integration friction), and find and manage their resources (access friction).

7 Design Challenges

In this section we examine the relationship between the environments' design aspects and the frictions developers experience (see Figure 7 for an overview). We describe how each of the environment's design aspects burden developers and offer design recommendations (denoted with a ☑) meant to better align the environment with developers' cross-application workflows.

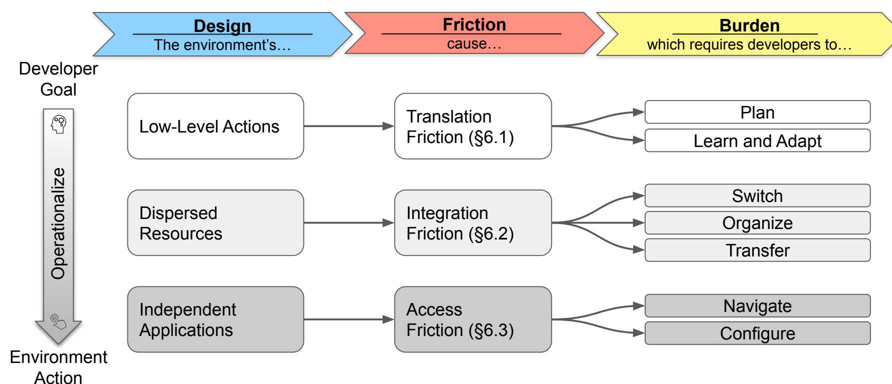

Fig. 7 Relationships between the environment's design and the burdens they impose on developers. Designs are ordered with burdens that are predominately cognitive at the top and predominately mechanical at the bottom to correspond with the process by which developers operationalize their goals into actions. For example, the environment's *low-level actions* cause *translation friction* which requires developers to *plan* their workflows and *learn and adapt* the available actions—primarily cognitive burdens.

7.1 Low-Level Actions

To be able to complete their goals, developers need to *plan* a workflow that consists of a series of low-level actions that the environment supports. Developers often resort to trial-and-error to construct these workflows, relying on feedback from their tools to *learn and adapt* their actions. These learned workflows are not always ideal, often incorporating actions from multiple applications, even when the goal can be achieved in a single application. Consolidating these workflows to use actions within a single application could help reduce the integration and access friction developers experience. Unfortunately, this requires developers to be aware of the relevant features within the application, and to invest time to learn how to translate their current workflows to use these features.

Use Familiar Features. Environments could observe developers' cross-application workflows and demonstrate equivalent workflows using features within the IDE (e.g., demonstrate the IDE's version control feature when a user commits their changes using shell commands).

While this approach could help developers become aware of more efficient actions for certain goals, it may not optimize the developer's overall workflow. It may also be difficult for developers to recall alternative workflows when they are needed if they are demonstrated after the developer has already completed their goal. A more ambitious approach would infer the developer's goal and automatically plan and execute the necessary actions when needed.

7.2 Dispersed Resources

Developers' cross-application workflows cause information to be shown in different places in the environment. This dispersion of information can make it difficult for developers to integrate information related to their goals. While developers are good at recognizing when and where they need to use this information, the process of obtaining the information is tedious and often disruptive as developers have to *switch* away from their current resource to manually obtain and *transfer* the information. One scenario developers frequently encounter is when they needed to include a specific resource identifier (e.g., an issue number) when entering text (e.g., a commit message). For example, including the issue number in the commit message requires developers to locate and switch to the issue on GitHub, locate the issue number, and transfer the number into their message.

Provide Information On-Demand. Environments could provide developers with goal-relevant auto-completion showing automatically extracted information (e.g., issue numbers, docker/commit SHAs, test names, file paths, and method names) directly where the developer is typing.

While providing information to developers where it is needed could reduce the friction associated with switching between resources, too much information could distract or overload developers. Careful consideration would need to be given to the way in which the information is provided to the developer.

7.3 Isolated Applications

Developers rely on a diverse set of tools across multiple isolated applications. Unfortunately, this isolation makes it difficult for developers to maintain a consistent state (e.g., working location, branch, environment variables) because they need to repeatedly *configure* this state in each application they use. Developers have to familiarize themselves with the different organizational structures and navigational mechanisms each application uses to represent and manage their state. Developers also need to manage state changes which occur inadvertently as they perform their actions. For example, it can be difficult to return to a resource once a developer closes it as they often have to re-navigate from an arbitrary location defined by the application.

> *Set Projects Globally.* Environments could explicitly track the active project and provide hints to allow all applications to open to that project automatically (e.g., IDEs, shells, GitHub, etc.) as a sensible default.

While setting the active project at the environment level could reduce the amount of navigation developers need to perform when switching between projects, opening applications to specific projects could be confusing. Providing an explicit "switch project" affordance to the environment could make this behaviour more obvious and would provide a hook enabling the environment to automatically set the appropriate state.

8 Discussion

In this section we compare the workflows used by developers across the two studies, describe the relationship of our findings to Information Foraging Theory (IFT), and discuss threats to validity.

8.1 Similarities and Differences Between Studies

We observed some notable similarities and differences in the types and frequency of frictions developers experienced in the user and field studies. In both studies, developers transferred content between resources a similar number of times. Both groups relied heavily on the shell to complete their tasks and experienced frictions associated with manually obtaining goal-relevant information to overcome the lack of information provided by individual shell

commands (e.g., explicitly listing the files in a directory or staged to be committed). Browsers were also used similarly between both groups to view documentation, manage their projects on GitHub, and test their web applications. However, subjects in the field study also used the browser to access other code-related services like build environments and cloud platforms, motivating further research into how these disparate services affect the development process.

Participants in the user study accumulated more windows over the course of the task as they followed links in the documentation and opened new shells. They also spent more time organizing these windows, likely the result of the smaller laptop screen and the number of windows they opened. Despite this organization, they still mis-switched frequently and resorted to either stepping through tabs individually or "thrashing" between windows seemingly randomly. We speculate the accumulation of windows was due to participants' unfamiliarity with the project and task: they may have been afraid to close resources that might be needed in the future, perceiving them as costly to re-open.

Subjects in the field study worked with smaller sets of windows, reusing the ones they had already opened (i.e., opening a link in the same tab or stopping a command in the shell to run another command) and were more willing to close resources. However, they spent more time obtaining goal-relevant information, for example, when trying to understand the behaviour of their programs or configuring their projects. They also spent more time managing their environment (e.g., handling updates, installing project dependencies and tools, setting environment variables). These are indicative of the fundamental costs of working in a cross-application environment and likely did not occur during the controlled user study due to the constrained nature of the task. In the field study, these frictions arose throughout the development process and subjects had to interrupt their tasks to handle them.

We believe that the observations from both studies are equally informative. The user study participants represent both novice and experienced developers who are working on new projects with unfamiliar tools and environments, requiring support to efficiently acquire and organize knowledge about their project. In contrast, the field study showed that developers familiar with their project and environment still need support to obtain information and manage state across the different applications and services required during the development process. The fact that we observed the similar frictions in both studies, even when developers had optimized their environments, suggests that the frictions generalize across developer experience, tools, and environments indicating the need for better support.

8.2 Information Foraging Across Applications

Piorkowski et al (2016) examined developers' foraging behaviour during a bug fixing task and identified six factors that made between-patch foraging diffi-

cult in the Eclipse IDE. We observed three of these factors in cross-application environments: disjoint topologies, scattered information prey, and misleading cues (false advertising). Consistent with this prior work, we found that getting to resources was costly due to the *disjoint topologies* used across applications forcing developers to navigate through different organizational structures (Section 6.3.1). However, we found this problem to be even more costly in cross-application environments because developers have to navigate from a base location (e.g., home directory) to their project in each application they use even before they can start looking for their desired information. As in the IDE, this information ends up *dispersed* across tabs and windows forcing developers to switch between them. However, these tabs can be open in one of many applications which can cause developers to mis-switch or walk through each tab and visually scan its contents (Section 6.2.1). This suggests that the cues provided in cross-application environments about the information contained in a window can also be insufficient or *misleading*.

In some cases, developers attempted to organize their windows anticipating revisits (Section 6.2.2). This is similar to *producer* effect proposed by Ragavan et al (2021) to describe the effort developers put into structuring their commits for future consumption by other developers working on the project. Surprisingly, we observed this even for resources managed entirely by the same developer and even for short-lived windows. Failing to anticipate these future needs led to expensive foraging to re-find resources.

Developers also adapted their environment to obtain-goal relevant information (Section 6.1.1) which is known as *enrichment* in IFT. Unlike prior work which considers enrichment as an isolated activity (e.g., filtering a set of links), we found that it is often a complex process requiring developers to learn or customize actions (Section 6.1.2) and to manage (Section 6.3.2) and transfer (Section 6.2.3) the information they generate. In particular, we found that when developers were entering unstructured text (e.g., a commit message), they often had to interrupt themselves to seek out information from a structured information source (e.g., a list of issue numbers) which often required developers to perform multiple actions across tools, consistent with the observations made by Ko et al (2007).

8.3 Threats to Validity

The nature of the experiments, observations, and analyses of this work gives rise to several threats to the validity of our findings.

Construct Validity. In the user study, we provided participants with a computer that had the environment pre-configured with the default install of the tools required for the project they were working with. This was to reduce the overhead associated with configuring the project and tooling, along with the privacy concerns associated with participants accidentally opening sensitive content on their own computers while being observed. This meant that the

tools and their configuration may have been different than those they were familiar with. Participants worked on a single task from an open source project. We chose to use a single task so we could compare participants' workflows with the ideal workflow using a diverse set of tools necessary to complete a real, moderately-complex task, within a reasonable amount of time.

In the field study, the dataset consisted of publicly streamed software development sessions. While Alaboudi and LaToza (2019) found that streamed sessions are not rehearsed and include actions needed to contribute to real projects, the sessions in our dataset may not represent the actual work done by developers working in a professional context. To reduce this threat, we discarded candidate videos which did not meet our screening criteria to remove streams that did not seem to match actual development sessions.

Internal Validity. As with all qualitative studies, our findings are subject to the researchers' perceptions. To mitigate this bias, multiple coders were involved in segmenting the transcripts and identifying developers' goals using multiple cues from participants. The three authors conducted multiple rounds of thematic analysis to refine the friction categories. However, it is possible that other researchers may have identified alternative frictions from the data.

In both studies, the developers knew they were being observed. This may have affected how they completed their tasks. In particular, participants in the user study may have felt they needed to complete the task quickly and rushed through the actions in an ad hoc manner. Subjects in the field study may have altered their actions since they knew their activities would be visible to an external audience. However, their experience presenting also helped to mitigate observation effects.

External Validity. It is possible that our results do not generalize to all developers working in all environments. In the user study we recruited developers through recruitment emails and personal contacts which may have introduced a selection bias. In the field study we collected a pool of candidate videos from developers listed on the Live Coders Team site and videos suggested by Twitch and YouTube. It is possible that by starting from a curated listing of streamers we selected videos that are not representative of streamers in general. To mitigate this, we included four streamers that were not part of the Live Coders Team. It is also possible that developers who choose to stream their development sessions do not represent developers generally.

In the user study, participants worked on a single task from a single project. While we selected a task from a project with a medium-sized code base in a popular language using conventional tools, it is possible that we may have identified different types of friction if we observed participants completing their own work tasks or additional tasks for other projects. While the field study mitigates this threat somewhat as those developers were working with a diverse set of projects and languages, the tasks they selected to record may not be representative of all development tasks.

Ultimately, we tried to make our results as generalizable as possible by observing 27 developers, with diverse backgrounds and experience, working on real development tasks. They were using typical desktop environments and a broad set of tools and applications. To improve the ecological validity of our findings, we only included frictions we observed across multiple developers and from both studies.

9 Conclusion

Developers rely on their tools and environments to complete their tasks. While new and more capable tools continue to be introduced to support increasingly complex development activities, the glue between these tools has largely remained unchanged. In this work we have examined how developers complete their tasks, finding frequent misalignment between developers and their environments. By watching 17 developers perform a controlled task and 10 industrial developers perform their own tasks, we observed developers frequently decomposing their tasks into high-level goals and then operationalizing those goals as sequences of low-level actions that they could then manually perform. The misalignment between the high-level goals the developers want to perform and the low-level actions provided by their environments induces friction that impedes their progress. We identify seven specific ways these frictions impact developers and provide design recommendations that could improve developer workflows and make them more productive.

Funding This work is supported, in part, by the Natural Sciences and Engineering Research Council of Canada grant no. PGSD3-519053-2018.

References

Adeli M, Nelson N, Chattopadhyay S, Coffey H, Henley A, Sarma A (2020) Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place. In: Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp 1–10, DOI 10.1109/VL/HCC50065.2020.9127264

Alaboudi A, LaToza TD (2019) An Exploratory Study of Live-Streamed Programming. In: Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp 5–13, DOI 10.1109/VLHCC.2019.8818832

Amann S, Proksch S, Nadi S, Mezini M (2016) A Study of Visual Studio Usage in Practice. In: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol 1, pp 124–134, DOI 10.1109/SANER.2016.39

Bannon L, Cypher A, Greenspan S, Monty ML (1983) Evaluation and Analysis of Users' Activity Organization. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 54–57, DOI 10.1145/800045.801580

Bao L, Xing Z, Xia X, Lo D, Hassan AE (2018) Inference of Development Activities from Interaction with Uninstrumented Applications. Empirical Software Engineering (ESE) 23(3):1313–1351, DOI 10.1007/s10664-017-9547-8

Bardram JE, Jeuris S, Tell P, Houben S, Voida S (2019) Activity-centric Computing Systems. Communications of the ACM 62(8):72–81, DOI 10.1145/3325901

Begel A, Khoo YP, Zimmermann T (2010) Codebook: Discovering and Exploiting Relationships in Software Repositories. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 125–134, DOI 10.1145/1806799.1806821

Beller M, Gousios G, Panichella A, Zaidman A (2015) When, How, and Why Developers (Do Not) Test in Their IDEs. In: Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 179–190, DOI 10.1145/2786805.2786843

Bernstein MS, Shrager J, Winograd T (2008) Taskposé: Exploring Fluid Boundaries in an Associative Window Visualization. In: Proceedings of the Symposium on User Interface Software and Technology (UIST), pp 231–234, DOI 10.1145/1449715.1449753

Bragdon A, Zeleznik R, Reiss SP, Karumuri S, Cheung W, Kaplan J, Coleman C, Adeputra F, LaViola JJ Jr (2010) Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 2503–2512, DOI 10.1145/1753326.1753706

Byström K, Hansen P (2005) Conceptual framework for tasks in information studies. Journal of the American Society for Information Science and Technology 56(10):1050–1061, DOI 10.1002/asi.20197

Chapuis O, Roussel N (2007) Copy-and-paste between overlapping windows. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 201–210, DOI 10.1145/1240624.1240657

Chattopadhyay S, Nelson N, Gonzalez YR, Leon AA, Pandita R, Sarma A (2019) Latent Patterns in Activities: A Field Study of How Developers Manage Context. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 373–383, DOI 10.1109/ICSE.2019.00051

DeLine R, Rowan K (2010) Code canvas: Zooming towards better development environments. In: Proceedings of the International Conference on Software Engineering (ICSE), vol 2, pp 207–210, DOI 10.1145/1810295.1810331

Dragunov AN, Dietterich TG, Johnsrude K, McLaughlin M, Li L, Herlocker JL (2005) TaskTracer: A Desktop Environment to Support Multi-tasking Knowledge Workers. In: Proceedings of the International Conference on Intelligent User Interfaces (IUI), pp 75–82, DOI 10.1145/1040830.1040855

Fritz T, Murphy GC (2010) Using information fragments to answer the questions developers ask. In: Proceedings of the International Conference on Software Engineering (ICSE), ICSE '10, pp 175–184, DOI 10.1145/1806799.1806828

Garrison DR, Cleveland-Innes M, Koole M, Kappelman J (2006) Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. The Internet and Higher Education 9(1):1–8, DOI 10.1016/j.iheduc.2005.11.001

González VM, Mark G, Mark G (2004) Constant, Constant, Multi-tasking Craziness: Managing Multiple Working Spheres. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 113–120, DOI 10.1145/985692.985707

Green TR (1989) Cognitive dimensions of notations. People and computers V pp 443–460

Henley AZ, Fleming SD (2014) The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 2511–2520, DOI 10.1145/2556288.2557073

Jeuris S, Houben S, Bardram J (2014) Laevo: A temporal desktop interface for integrated knowledge work. In: Proceedings of the Symposium on User Interface Software and Technology (UIST), pp 679–688, DOI 10.1145/2642918.2647391

Kaptelinin V (2003) UMEA: Translating Interaction Histories into Project Contexts. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 353–360, DOI 10.1145/642611.642673

Kersten M, Murphy GC (2005) Mylar: A Degree-of-Interest Model for IDEs. In: Proceedings of the International Conference on Aspect-oriented Software Development (AOSD), pp 159–168, DOI 10.1145/1052898.1052912

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. IEEE Transactions on Software engineering 32(12):971–987, DOI 10.1109/TSE.2006.116

Ko AJ, DeLine R, Venolia G (2007) Information Needs in Collocated Software Development Teams. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 344–353, DOI 10.1109/ICSE.2007.45

LaToza TD, Venolia G, DeLine R (2006) Maintaining Mental Models: A Study of Developer Work Habits. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 492–501

Lawrance J, Bellamy R, Burnett M (2007) Scents in Programs:Does Information Foraging Theory Apply to Program Maintenance? In: Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp 15–22, DOI 10.1109/VLHCC.2007.25

Lawrance J, Bogart C, Burnett M, Bellamy R, Rector K, Fleming SD (2013) How programmers debug, revisited: An information foraging theory perspective. IEEE Transactions on Software engineering 39(2):197–215, DOI 10.1109/TSE.2010.111

Lewis C, Wharton C (1997) Chapter 30 - Cognitive Walkthroughs. In: Helander MG, Landauer TK, Prabhu PV (eds) Handbook of Human-Computer Interaction, 2nd edn, pp 717–732, DOI 10.1016/B978-044481862-1.50096-0

Maalej W (2009) Task-First or Context-First? Tool Integration Revisited. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp 344–355, DOI 10.1109/ASE.2009.36

Minelli R, Mocci A, Lanza M (2015a) I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp 25–35, DOI 10.1109/ICPC.2015.12

Minelli R, Mocci A, Lanza M (2015b) The Plague Doctor: A Promising Cure for the Window Plague. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp 182–185, DOI 10.1109/ICPC.2015.28

Murphy GC, Kersten M, Findlater L (2006) How Are Java Software Developers Using the Eclipse IDE? IEEE Software 23(4):76–83, DOI 10.1109/MS.2006.105

Oliver N, Smith G, Thakkar C, Surendran AC (2006) SWISH: Semantic Analysis of Window Titles and Switching History. In: Proceedings of the International Conference on Intelligent User Interfaces (IUI), pp 194–201, DOI 10.1145/1111449.1111492

Oliver N, Czerwinski M, Smith G, Roomp K (2008) RelAltTab: Assisting Users in Switching Windows. In: Proceedings of the International Conference on Intelligent User Interfaces (IUI), pp 385–388, DOI 10.1145/1378773.1378836

Pilzer J, Rosenast R, Meyer AN, Huang EM, Fritz T (2020) Supporting Software Developers' Focused Work on Window-Based Desktops. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 1–13

Piorkowski D, Fleming SD, Scaffidi C, Burnett M, Kwan I, Henley AZ, Macbeth J, Hill C, Horvath A (2015) To fix or to learn? How production bias affects developers' information foraging during debugging. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 11–20, DOI 10.1109/ICSM.2015.7332447

Piorkowski D, Henley AZ, Nabi T, Fleming SD, Scaffidi C, Burnett M (2016) Foraging and Navigations, Fundamentally: Developers' Predictions of Value and Cost. In: Proceedings of the Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE), pp 97–108, DOI 10.1145/2950290.2950302

Piorkowski DJ, Fleming SD, Kwan I, Burnett MM, Scaffidi C, Bellamy RK, Jordahl J (2013) The whats and hows of programmers' foraging diets. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 3063–3072, DOI 10.1145/2470654.2466418

Pirolli P, Card S (1995) Information foraging in information access environments. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 51–58, DOI 10.1145/223904.223911

Ragavan SS, Codoban M, Piorkowski D, Dig D, Burnett M (2021) Version Control Systems: An Information Foraging Perspective. IEEE Transactions

on Software engineering 47(8):1644–1655, DOI 10.1109/TSE.2019.2931296
Rattenbury T, Canny J (2007) CAAD: An Automatic Task Support System. In: Proceedings of the Conference on Human Factors in Computing Systems (CHI), pp 687–696, DOI 10.1145/1240624.1240731
Robertson G, Horvitz E, Czerwinski M, Baudisch P, Hutchings DR, Meyers B, Robbins D, Smith G (2004) Scalable Fabric: Flexible Task Management. In: Proceedings of the Working Conference on Advanced Visual Interfaces (AVI), pp 85–89, DOI 10.1145/989863.989874
Robillard M, Coelho W, Murphy G (2004) How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Transactions on Software engineering 30(12):889–903, DOI 10.1109/TSE.2004.101
Robillard MP, Murphy GC (2007) Representing concerns in source code. ACM Transactions on Software Engineering and Methodology 16(1):3–es, DOI 10.1145/1189748.1189751
Roethlisberger D, Nierstrasz O, Ducasse S (2009) Autumn Leaves: Curing the Window Plague in IDEs. In: Proceedings of the Working Conference on Reverse Engineering (WCRE), pp 237–246, DOI 10.1109/WCRE.2009.18
Sillito J, Murphy GC, Volder KD (2008) Asking and Answering Questions during a Programming Change Task. IEEE Transactions on Software engineering 34(4):434–451, DOI 10.1109/TSE.2008.26
Singer J, Lethbridge T, Vinson N, Anquetil N (1997) An Examination of Software Engineering Work Practices. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), pp 21–36
Smith G, Baudisch P, Robertson GG, Czerwinski M, Meyers BR, Robbins DC, Andrews DB (2003) GroupBar: The TaskBar Evolved. In: Proceedings of the Australian Conference on Human-Computer Interaction (OZCHI), vol 3, p 10
Soloway E (1986) Learning to program = learning to construct mechanisms and explanations. Communications of the ACM 29(9):850–858, DOI 10.1145/6592.6594
Strauss A, Corbin J (1994) Grounded theory methodology. In: Denzin NK, Lincoln YS (eds) Handbook of qualitative research, Sage Publications, pp 273–285
Tashman C (2006) WindowScape: A Task Oriented Window Manager. In: Proceedings of the Symposium on User Interface Software and Technology (UIST), pp 77–80, DOI 10.1145/1166253.1166266
Vallacher RR, Wegner DM (2012) Action identification theory. In: Handbook of Theories of Social Psychology, Vol. 1, Sage Publications Ltd, pp 327–348, DOI 10.4135/9781446249215.n17
Venolia G (2005) Bridges between silos: A microsoft research project
Wang Y, Chiew V (2010) On the Cognitive Process of Human Problem Solving. Cognitive Systems Research 11(1):81–92, DOI 10.1016/j.cogsys.2008.08.003
Zhao S, Chevalier F, Ooi WT, Lee CY, Agarwal A (2012) AutoComPaste: Auto-completing text as an alternative to copy-paste. In: Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI), pp

365–372, DOI 10.1145/2254556.2254626

Čubranić D, Murphy GC (2003) Hipikat: Recommending Pertinent Software Development Artifacts. In: Proceedings of the International Conference on Software Engineering (ICSE), pp 408–418