2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Latent Patterns in Activities: A Field Study of How Developers Manage Context

Souti Chattopadhyay*, Nicholas Nelson*, Yenifer Ramirez Gonzalez*, Annel Amelia Leon*,
Rahul Pandita† and Anita Sarma*
*Oregon State University, Corvallis, OR, USA †Phase Change Software, Golden, CO, USA
{chattops, nelsonni, ramireye, leonan}@oregonstate.edu, rpandita@phasechange.ai, anita.sarma@oregonstate.edu

***Abstract*—In order to build efficient tools that support complex programming tasks, it is imperative that we understand how developers program. We know that developers create a context around their programming task by gathering relevant information. We also know that developers decompose their tasks recursively into smaller units. However, important gaps exist in our knowledge about: (1) the role that context plays in supporting smaller units of tasks, (2) the relationship that exists among these smaller units, and (3) how context flows across them. The goal of this research is to gain a better understanding of how developers structure their tasks and manage context through a field study of ten professional developers in an industrial setting. Our analysis reveals that developers decompose their tasks into smaller units with distinct goals, that specific patterns exist in how they sequence these smaller units, and that developers may maintain context between those smaller units with related goals.**

***Index Terms*—context, task decomposition, field study**

I. Introduction

Programming is a creative endeavor in which developers engage in different types of closely-related activities to complete a development task. They explore different solutions [1], review past versions [2], and backtrack [3]. We need to know how these activities relate, interleave, and build upon each other to truly understand how developers work. This is essential if we wish to build tools that seamlessly support all the different programming activities.

Studying developer activity has been a topic of research for decades. As early as 1984 Vorburgh et al. studied programming environments, identifying 14 factors influencing team productivity [4]. More recently Ko et al. studied the information needs of developers [5] and Meyer et al. studied how developers' work habits correspond with their perceptions of productivity [6].

From the collected work, two key observations emerge that are pertinent to the research questions explored in this paper.

First, developers work in short bursts of activities. Gonzalez and Mark [7] and Meyer et. al. [6] found that developers organized their development tasks into smaller, basic units of work.

Second, as developers work they create a context that drives their development activities. This context encompasses relevant information needed to complete the development task [8]. For example, in order to fix a bug, a developer needs to know about the bug (from its description in the issue tracker), how to replicate the bug (discussion snippet in the issue tracker), what the code currently does (lines of source code), and so on. All these information elements together create the context of this bug-fix task.

These two observations have motivated several lines of research. Researchers have built tools to recommend the project artifacts that are relevant to a developer's task [9]–[11]. Other work has looked further into how developers "recursively decompose a development task into a hierarchy of subtasks" [12]. Additionally, researchers have investigated how developers' working style across subtasks correspond with their perception of productivity [6], and how productive developers' differentiate themselves based on how they decompose their testing tasks [13].

Despite all this work, important gaps still remain in our understanding of how developers go about solving complex programming tasks. Specifically, while research has identified the role that context plays in supporting development tasks and that developers decompose their tasks into smaller units, open questions remain. What is the relationship among these smaller units of work? How does context impact smaller units of tasks? And how does context flow from one unit to another?

While it is useful to study individual smaller units of tasks, it is equally useful to understand how these units relate to one another. For instance, do patterns exist in how developers structure their development tasks into smaller units? And do these correspond to specific types of development task (e.g, bug fix, refactoring code, implementation)? And, if patterns do exist, how does context evolve from one smaller task to another? Is context tossed away upon the completion of a smaller task and constructed anew for the next? Is it carried forward in time?

In this paper, we aim to close this gap by observing developers in a field study. We observed ten developers at a software company. We recorded their development activities, which included interactions with artifacts while working on their own development tasks, and in their own programming environment. We conducted a follow-up survey which validated our findings on how developers decompose tasks and create context.

This study makes the following observations about development behavior:

- Different patterns emerge in how developers organize their tasks into smaller units.

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00051

- Patterns in organization of smaller units depend on the type of the development task.
- Goals of smaller units are instrumental in creating context, which guides developers' interactions with artifacts.
- Depending on the relationship between smaller units, developers maintain (or drop) the context as they shift between units.

Understanding these development behaviors are fundamental to tool builders and researchers when building development environments that better support programming activities.

II. Definitions

Studies of developer activity require a concrete set of terminology. In this section, we define key terms through an example scenario involving Charlie (a programmer).

Goal: *The end towards which development effort is directed.*

For example, during a normal workday, Charlie has a goal to refactor a portion of the code base.

Subgoal: *Goals are composed of subgoals, which represent smaller actionable objectives.*

For example, to refactor the code base, Charlie can have subgoals: understand why the code has to be refactored (Subgoal 1), identify the parts of the code that have to be refactored (Subgoal 2), and replace old code with new code (Subgoal 3).

Figure 1 shows an example of a goal that is broken down into three subgoals. A developer may identify the subgoals right away, or create them organically as they proceed.

Action: *A development effort comprised of steps performed by a developer towards reaching their subgoal.*

For example, to replace old code with new code (Subgoal 3), Charlie may take the following actions: open the class file A in his editor (navigate), read file A (read), delete old extraneous sections of code (edit), add new code (edit), and compile the resulting code (execute). See Table II for specific definitions of these actions.

Episode: *A series of loosely connected actions that are performed continuously one after the other (temporally related) or performed to complete a subgoal.*

For example, the above actions compose an episode guided by Charlie's subgoal of replacing old code with new code (Subgoal 3). This is shown in Episode 4 of Figure 1. Had Charlie been interrupted or taken other actions related to another subgoal, the above set of actions would be represented as two separate episodes.

Context: *The information that is used to perform a development effort, which includes the artifacts, interactions with the artifacts (observable), and sensemaking of the artifacts (not observable).*

For example, Charlie reads lines of code (artifact) to understand where to refactor incorrect pieces of code. The code artifact and the interaction with that artifact (reading) comprise the context. Similarly, Charlie may delete the "incorrect" code before writing new code. Although the artifact remains the same, different contexts are created based on Charlie's sensemaking approach.

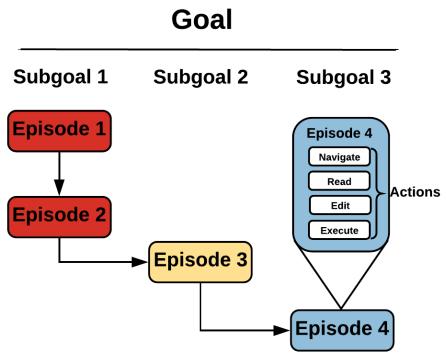


Fig. 1: Depiction of a Goal, subgoals, episodes, and actions.

Working Context: *The specific context that is required to complete actions in an episode.*

Developers may use parts (or all) of a working context from a prior episode when executing their actions in the current episode. For example, Charlie reads a portion of the code and uses that working context to guide his refactoring of the code.

III. Methodology

We conducted a field study where we observed ten developers working on their programming tasks. Each session was approximately one hour long and included a 15–minute retrospective interview (total observation time: 6 hours, 40 minutes). We unobtrusively observed the workspace, computer-screens, and interactions of each developer following the "fly on the wall" technique described by Preissle and Grant [14]. We then validated our findings through a follow-up survey, per guidelines by Easterbrook et al. [15].

A. Study Design

Study participants were recruited from a US-based software startup. This startup operates in the areas of distributed developer tools and services, which includes: program analysis, UI, infrastructure/middleware support, and tool R&D. The variety of topics allowed for diverse languages and working styles among our participants. Participants volunteered to take part in this study and were not compensated.

TABLE I: Study Participant Demographics

Ptc.[i]	Gender	Exp.[ii]	Language(s)[iii]	Editor[iv]
P1	M	21y 0m	Java	Eclipse
P2	M	1y 11m	Clojure	Eclipse
P3	M	1y 10m	Clojure, Java	Emacs
P4	M	7y 3m	Clojure, Python	Emacs
P5	M	2y 0m	Clojure, Java, Haskell	Emacs
P6	M	2y 0m	TypeScript, Java, Clojure	VS Code
P7	M	5y 0m	C/C++	Emacs
P8	F	15y 0m	JavaScript, CSS	VS Code
P9	M	0y 9m	C, Prolog	Sublime
P10	F	1y 0m	Python	PyCharm

[i] Ptc. = Participant [ii] Exp. = Years/months of software development experience [iii] Preferred programming language(s) [iv] Editor used in session

Table I presents participant demographic details, including gender, software development experience, their preferred programming language(s), and preferred editor. The average was 5 years, 9 months of software development experience.

We observed developers performing their regular development tasks on a typical workday. We demonstrated the think-aloud protocol [16] and requested participants to verbalize their thoughts and interactions during the session, which we recorded using two separate microphones for redundancy. Each session was recorded using screen capture software. Additionally, we video recorded participants' physical workspace to capture all artifacts (e.g. paper and whiteboard media).

For each session, one researcher was positioned behind the participant taking in-situ field notes. This researcher noted information about activities performed by the participant (e.g. reviewing design notes on paper before writing code in the editor). An additional researcher was located in a separate room not visible to the participant. This researcher monitored the screen and audio recording of the participant to take secondary field notes. The first and second authors were responsible for the data collection and alternated between primary and secondary positions.

Sessions were time-boxed to 45 minutes, which allowed time to perform a retrospective interview (15–20 minutes). Longer sessions were not feasible due to time restrictions at the startup. Sessions stopped when participants completed their task, which resulted in some sessions being shorter than 45 minutes in duration.

At the end of the session, participants were asked to complete a brief demographic survey (see Table I for responses). While participants completed the survey, the researchers compared their field notes to identify portions of the session that were of interest. Clarifying questions about the session were asked in a 15–minute retrospective interview, which included replaying portions of the screen capture video in order to refresh participants' memory.

B. Data Analysis

After completing all study sessions, the retrospective interviews and think-aloud verbalizations were transcribed.

Coding of the transcribed data was performed in three steps. First, each *action* was coded by multiple raters and a high inter-rater reliability (IRR) measures was maintained. Second, the units of measurement—*subgoals* and *episodes*—were defined through negotiated agreement among the first four authors using the first half of session data collected from P4. Third, the *patterns* were identified by analyzing the units of measurement across the entire dataset using negotiated agreement. A detailed description and walk-through of the coding scheme can be found on our companion site[1].

Coding of the session data (screen capture video, transcripts, and field notes) included: (1) coding the verbalized overall *goal* and *subgoals* within the session, (2) annotating the active artifacts (i.e. artifacts which participants interacted with), and (3) *actions* taken by the participant. These *actions* are defined in Table II.

[1]https://sarmaresearch.github.io/ICSE19-LatentPatternsInActivities/

TABLE II: Action Codes

Action	Definition
Read	Examining information from artifacts (e.g. code, documentation, terminal output)
Edit	Any change made directly to code or related artifacts.
Navigate	Moving within or among artifacts (e.g. pulling files from Git, opening files, scrolling through a file).
Execute	Compiling and/or running code.
Ideate	Constructing mental model of future changes.

Two five minute segments of session data collected from P4 were unitized into 24 and 23 *actions*. The first four authors individually coded each *action* with the codes described in Table II. We calculate inter-rater reliability (IRR) using Fleiss' Kappa across all four raters. The kappa value was 0.647 (24 instances, 4 raters, $p-value < 0.001$) and 0.908 (23 instances, 4 raters, $p - value < 0.001$) in first and second rounds of coding, respectively.

We identified the *subgoals* through a combination of coded *actions* and verbalizations. *Actions* were segmented to their appropriate *subgoal* based on the timestamp and the thematic topic of each *subgoal*. For most segments, the subgoal was readily discernible from the verbalizations. For example, P4 indicated "*I need to create a new entry into this module.*" This verbalization occurred at the same time that P4 switched from the *read* action to the *edit* action, which provided further evidence of the transition between subgoals.

Additionally, we identified the *episodes* of actions within the data by using the actions, subgoals, and relationships between them. The transitions between episodes were identified based upon either a change in subgoal, or a period of non-development activity (e.g. interruptions or taking a break). For example, P6 mentioned: "*Now I am going to move node methods from the hierarchy service.*" In this instance, the verbalization provided a clear delineation between episodes based on the described subgoal (*moving node methods*).

To ensure validity of episode and subgoal coding, the first four authors incrementally coded the first half of session data from P4 using *negotiated agreement*. Negotiated agreement was used to create a standardized coding scheme and improve it to an acceptable point where there was no ambiguity in any of the data points [17]. After reaching agreement, the first and second authors coded the episodes and subgoals for participants P1–P4, and the third and fourth authors coded the data for participants P5–P10.

C. Validation Survey

To validate the patterns of episodes during data analysis, we conducted a follow-up survey with participants. We were unable to contact P8 since she had left the company in-between the study and start of the survey.

The survey consisted of two sets of questions and took about 15 minutes to complete. The full set of survey questions can be found on our companion site[1].

The first set of questions introduced a brief development scenario and provided participants with a set of information elements (e.g., Java compile error message indicating reference error, Java compile error message indicating memory out of bounds) and multiple software development actions (e.g., execute code in a terminal window). Participants had to match the information elements that they thought were relevant to the actions based on the description of the scenario.

The second set of questions showed patterns in which episodes could be arranged, which are described later in Section IV-B. These patterns were described in text as well as graphically. Participants had to rank the patterns based on how frequently they used them, and then provide rationale behind their most-frequently used and least-frequently used pattern.

D. Limitations of the study

As any field study, our findings are derived from a limited number of observations regarding the development efforts of our participants from a single software development company. However, our participants performed different types of development tasks, including implementing core program analysis, server management, web interface development, and analysis modeling. They also used varied environments such as feature-rich IDEs (e.g. Eclipse) to advanced code editors (e.g. Emacs). Our observational study intends to identify patterns that arise in common software development tasks. Generalizability, although desirable, was not a primary objective of our study. Instead we aim to present findings that can be transferred to various environments, providing contextual support to programmers [18].

While we only observed 10 developers for 45 minutes each, our primary units of analysis are the 242 episodes and 130 subgoals discovered during these sessions. The inherent nature of an observational think-aloud study might cause the data to be affected by the Hawthorne effect, response bias or create additional cognitive load in participants needing to think aloud [19]. Such limitations are prevalent in protocol studies and can be removed in future studies that instrument a developers' workspace in the background.

As is the case in any qualitative study, our findings are subjective to the researchers' perceptions. We mitigated this threat through rigor in our analysis process by using four raters, maintaining inter-rater reliability and using a well defined coding scheme [20]. We also validated our findings with the perceptions of the developers through a follow-up survey.

IV. Results

A. Goal Structuring

RQ1: *How do developers structure their software development goals?*

Developers structure their development goals into episodes of actions, each of which is driven by a particular subgoal. We observed 242 episodes driven by 130 distinct subgoals. Table III presents the individual number of episodes and subgoals per participant during the study, the duration (in `hh:mm:ss` format) of each session, and the goal type for the session (as verbally indicated by the participant at the start of each session).

TABLE III: Goals, Episodes, and Subgoals in Field Study

Participant	Goal Type	Episodes	Subgoals	Duration[i]
P1	Debugging	38	10	0:46:43
P2	Implementing	23	12	0:47:36
P3	Refactoring	24	17	0:46:23
P4	Implementing	15	9	0:43:37
P5	Debugging	37	12	0:30:40
P6	Refactoring	10	8	0:34:06
P7	Refactoring	15	11	0:28:23
P8	Implementing	22	14	0:44:09
P9	Implementing	27	17	0:45:19
P10	Debugging	32	20	0:34:02
	Totals	242	130	6:40:58

[i] Duration of each study session (`h:mm:ss`)

On average study sessions were 40 minutes 6 seconds long, and included 24.2 episodes driven by 13 subgoals. The distribution of the episodes and subgoals across goal types reveal that not all goal types use the same structure.

Participants working on an *Implementation* goal (four participants) had an average study duration of 45 minutes, 10 seconds; with 21.75 episodes driven by 13 subgoals. The average time spent per episode was 2 minutes 5 seconds, whereas the average time spent per subgoal was 3 minutes 28 seconds.

Participants working on a *Refactoring* goal (three participants) had an average study duration of 36 minutes 17 seconds; with 16.33 episodes driven by 12 subgoals. The average time spent per episode was 2 minutes 13 seconds, whereas the average time spent per subgoal was 3 minutes 1 second.

For participants working on a *Debugging* goal, they deviated from the overall average. There were three participants working on Debugging goals and the average study duration was 37 minutes 8 seconds; with 35.67 episodes driven by 14 subgoals. The average time spent per episode was 1 minute 2 seconds, whereas the average time spent per subgoal was 2 minutes 39 seconds.

We found that Debugging goals require shorter, more frequent episodes of actions. Although the overall session was similar, participants working on a Debugging goal worked in more episodes when compared to participants working on an Implementation or Refactoring goal, with 13.92 and 19.34 additional episodes respectively. The average duration of the episodes for the Debugging goal was also shorter than those of Implementation or Refactoring goals, 1 minute 2 seconds and 1 minute 11 seconds shorter respectively.

We hypothesize that debugging requires more cognitive load per episode of actions. Debugging requires locating the source of the bug, comprehending the code associated with the bug, and modifying the code to fix it. Whereas, Implementation and Refactoring actions rely more on sensemaking and modifying the code than finding specific parts of faulty code.

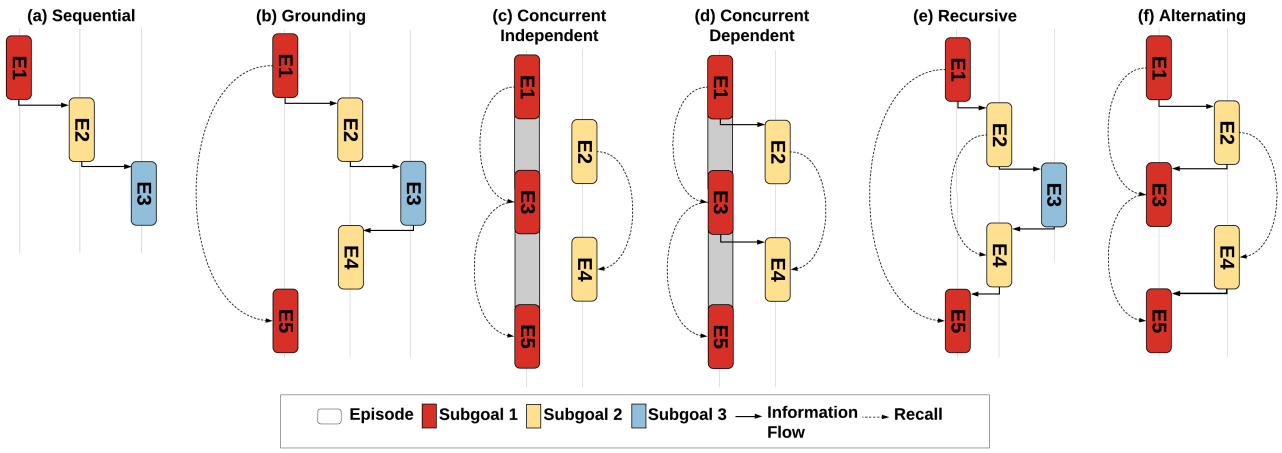


Fig. 2: Visual Illustration of Observed Patterns

The average subgoal duration for Debugging was also the shortest (2 minutes, 39 seconds), whereas Implementation and Refactoring sub-goals were 49 seconds and 22 seconds longer, respectively. This indicates that Debugging goals require more frequent context-switching between subgoals.

Gonzalez et al. [7] found that people spend about 3 minutes on average per task (which is synonymous with a subgoal in our study). Our results also show that subgoals last for 3 minutes 5 seconds on average. We additionally find that the average time per subgoal varies according to the goal type and that developers purposefully structure their goals in order to switch between subgoals approximately every 3 minutes.

In summary, we find that, on average, developers structure their goals into 1.87 episodes that are focused on the same subgoal for about 3 minutes 5 seconds. The duration and frequency of these subgoals and episodes vary according to the goal type.

B. Patterns in Episode Structuring across Subgoals

RQ2: *What patterns do developers employ when working in and across subgoals?*

Developers decompose their goals into subgoals, which drive episodes of actions. We observed that these episodes occur in patterns that are formed as a developer works through different subgoals over time.

Our participants structured episodes into five distinct patterns. We observed that participants arranged their episodes in sequence across different subgoals (*Sequential* pattern), concurrently across two subgoals (*Concurrent*), recursively into increasingly fine-grained subgoals (*Recursive*), by revisiting previous subgoals to reorient (G*rounding*), or by exploring two alternate subgoals (*Alternating*). Figure 2 illustrates each of these patterns.

*1) **Sequential Pattern**:* Developers work in sequential episodes that span different subgoals. Each subgoal helps define subsequent subgoals.

Figure 2 (a) illustrates P3 decomposing his refactoring goal into three sequential subgoals. Towards Subgoal 1, P3 replicated a method related to the `translator` property ensuring that there were no errors in the process. After successfully completing this subgoal, he replicated several methods related to the `conversion` property (Subgoal 2). Finally, he created a query method to be used in the replicated properties (Subgoal 3). Throughout this sequence, P3 did not return to working on any prior subgoal.

TABLE IV: Frequency of Pattern Instances

Ptc.[i]	Sequential	Concurrent	Grounding	Recursive	Alternate
P1	1	5	4	2	0
P2	2	1	1	1	0
P3	2	0	1	2	0
P4	1	0	2	1	0
P5	0	2	2	0	0
P6	2	0	1	0	0
P7	2	0	1	0	0
P8	3	0	2	0	0
P9	3	0	2	2	2
P10	4	1	3	1	0
Avg.[ii]	2.00	0.90	1.90	0.90	0.20

[i] Ptc. = Participant [ii] Average instances per participant for a given pattern.

The Sequential pattern was the most frequent pattern. We observed 20 instances of this pattern across 9 out of 10 participants (90%). This pattern was typically used during code experimentation. Table IV provides the frequency of pattern occurrences per participant.

Developers perceived the Sequential pattern to be useful and prevalent in their work. 88.9% of validation survey participants (8 out of 9) indicated that they use the Sequential pattern either *sometimes* or *most of the time*; see Table V for individual participant responses. Participants ranked the Sequential pattern as the most frequently used pattern (avg. rank: 1.55). P2 indicated:

"*Using this pattern makes it easier to decompose*

TABLE V: Pattern Validation through Member Checking

Ptc.[i]	Sequential		Concurrent		Grounding		Recursive		Alternate	
	Frequency	Rank	Frequency	Rank	Frequency	Rank	Frequency	Rank	Frequency	Rank
P1	Sometimes	2	Sometimes	4	Sometimes	3	Most of the time	1	Rarely	5
P2	Sometimes	1	Rarely	5	Sometimes	2	Sometimes	3	Sometimes	4
P3	Sometimes	2	Never	5	Most of the time	1	Rarely	3	Rarely	4
P4	Sometimes	2	Sometimes	3	Most of the time	1	Sometimes	4	Rarely	5
P5	Sometimes	2	Never	5	Rarely	1	Most of the time	3	Rarely	4
P6	Sometimes	1	Rarely	5	Sometimes	2	Sometimes	3	Rarely	4
P7	Sometimes	1	Sometimes	3	Sometimes	4	Sometimes	2	Sometimes	5
P9	Most of the time	1	Most of the time	2	Sometimes	4	Rarely	5	Rarely	3
P10	Rarely	2	Sometimes	5	Most of the time	1	Rarely	4	Sometimes	3
Avg.	Sometimes (1.6)[ii]	1.6	Never (4.1)[ii]	4.3	Sometimes (2.1)[ii]	1.9	Sometimes (3.1)[ii]	3.3	Rarely (4.1)[ii]	4.0

[i] Ptc. = Participant [ii] Frequency responses converted to numerical form for calculating averages (0: Never, 1: Rarely, 2: Sometimes, 3: Most of the time).

complex tasks because previous stages are completed before moving on to the next."

2) ***Grounding Pattern****:* Developers reorient themselves to their overarching goal by revisiting a prior subgoal. Grounding typically involves building and executing code, running tests, and checking system status; which are all time-intensive processes.

Figure 2 (b) illustrates P9 evaluating the data format of a query (Subgoal 1). He then edited the query (Subgoal 2), and created a method to wrap the output data (Subgoal 3). In order to ensure that the query was returning data in the correct format, he grounded himself by revisiting the original query (return to Subgoal 1). P9 had to recall the previous format of the query (episode 1) to evaluate whether his implementation was successful.

The frequency with which participants grounded themselves varied across participants. Eight participants (80%) grounded themselves after a single episode. For example, P9 explored different syntax to implement a sorting algorithm. He built and executed the code at several points to verify his solution. These executions represent instances of the Grounding pattern, which typically occurred when participants iterated through experimental solutions. We also observed four participants (40%) working through several episodes before grounding themselves. In these cases, participants typically grounded to evaluate their progress, reorient to the overarching goal, and transition to the next subgoal. For example, P1 grounded himself after six episodes and said:

> *"What I wanna try getting next is just to [implement another minor feature]"*

88.8% validation survey responses (8 of 9) indicated that participants use the Grounding pattern either *sometimes* or *most of the time*. Participants ranked the *Grounding* pattern as the second most frequently used pattern (avg. rank: 2.11).

We conclude that when developers work on exploratory subgoals they evaluate their progress by "grounding". P4 said:

> *"[Grounding pattern] allows me to see the results of each step (like stepping through in a debugger)..."*

3) ***Concurrent Pattern****:* Developers occasionally work concurrently towards one subgoal, while waiting on a process relating to a different subgoal (e.g. builds, tests, code reviews).

Based on the relationships between subgoals, we observed two variations of the *Concurrent* pattern: independent and dependent concurrency. ***Concurrent Independent*** pattern instances emerge when the two concurrent subgoals are independent of each other (i.e. the completion of one subgoal is not required in order to complete the other subgoal). ***Concurrent Dependent*** pattern instances emerge when the two concurrent subgoals are dependent upon each other (i.e. a deviation in the expected behavior of processes in one subgoal require attention in the other subgoal).

Figure 2 (c) illustrates P5 triggering a build (Subgoal 1). While waiting for the build to finish, he began searching and reading documentation about a specific query API (Subgoal 2). When the build finished with an error notification, he switched back to the build configuration and made additional changes and triggered a new build (Subgoal 1). Instead of waiting for the build, P5 again returned to reading documentation about the query API (Subgoal 2). This is an occurrence of the *Concurrent Independent* pattern. During the study, P5 described this sequence of episodes by saying:

> "*While this is running, I am going look for the query slice... I haven't interacted with the query interface in a couple months so I am going to familiarize myself with the code.*"

By contrast, the *Concurrent Dependent* pattern is illustrated in Figure 2 (d), which shows P1 running the tests (Subgoal 1) and starting to commit all modified files (Subgoal 2) prior to seeing the test results. Since he was working concurrently, P1 had already committed several files before some of the tests failed. This forced P1 to halt committing, fix the broken code, and restart the tests. P1 then had to revert his previous commits (Subgoal 2), and exclaimed: *"Now, I gotta get rid of those [commits]!"* Committing the modified files (Subgoal 2) was dependent upon the tests passing (Subgoal 1). We observed nine instances of this pattern across 40% participants (4 out

of 10).

55.6% survey participants (5 out of 9) stated that they use the *Concurrent* pattern either *sometimes* or *most of the time*. However, two participants indicated that they *never* use this pattern. Participants ranked the *Concurrent* pattern as the least frequently used (avg. rank: 4.11).

The episodes directed towards one of the subgoals in the *Concurrent* pattern are off-loadable either to computer processes or other personnel, which might explain the low frequency with which participants use this pattern.

As P6 said: *"[I will use the Concurrent pattern] if one of my tasks involves a lot of waiting or downtime."* Additionally, a blocked subgoal can be attributed to coordination issues such as waiting for a code review, as indicated by P3:

> *"[I will use the Concurrent pattern] when I have two subtasks completely unrelated and I'm waiting for code review in some of them."*

Past work has shown that developers pursue two subgoals concurrently to increase productivity [6]. We similarly find that working concurrently on *independent* subgoals can be beneficial to a developers' productivity. However, we see that working concurrently on *dependent* subgoals can actually inhibit productivity and force unwanted context-switches.

4) ***Recursive Pattern****:* Developers decompose their current subgoal into recursively nested subgoals. In this pattern, each subgoal is dependent upon successive subgoal(s).

Figure 2 (e) illustrates P4 attempting to create a query function. He initially started to debug the existing query code (Subgoal 1). After realizing that a helper function was needed (Subgoal 2), he devised that an appropriate filter function was needed during implementation. After successfully implementing the filter (Subgoal 3), he retraced back and completed the helper function (Subgoal 2) and query (Subgoal 1).

60% of participants (6 of 10) used the *Recursive* pattern. This pattern typically emerged when developers were working on a relatively unexplored problem space.

66.7% of validation survey participants (6 out of 9) stated that they *sometimes* or *most of the time* recursively structure their subgoals. Recursive pattern was perceived as the third most frequently used pattern (avg. rank: 3.11).

The variation in the participants' ranking of the Recursive pattern suggest that the situations in which this pattern can be used are infrequent or complex. P9 stated:

> "*...I would use it if I was facing a problem that I anticipate to be very large and complex.*"

5) ***Alternating Pattern****:* Developers occasionally work simultaneously on two or more subgoals that represent alternative solutions to a larger goal. Developers frequently switch between episodes towards different subgoals. Unlike the *Concurrent* pattern, in an *Alternating* pattern developers maintain some part of their context toward the active subgoal at all times.

Figure 2 (f) illustrates P5 exploring two alternate solutions to a missing dependency error. P5 started implementing the solution from a forum posting (Subgoal 1), followed by implementing a different solution from another posting (Subgoal 2). P5 continued switching between the two subgoals, iteratively completing portions of both solutions and comparing.

We observed two instances of this pattern from one participant (10%), thus making this pattern the least frequent in our study.

77.8% of validation survey participants (7 of 9) indicated that they *rarely* use the *Alternating* pattern. Participants also ranked the *Alternating* pattern as the least frequently used pattern (avg. rank: 4.11). The *Alternating* and *Concurrent* patterns share the position of least frequent.

Alternately exploring solutions requires maintaining different contexts, as pointed out by P7:

> "*I use this pattern as a way to consider the context of a problem across different solutions.*"

The *Alternating* pattern occurred rarely in our study, partially due to the sessions being limited to one overarching goal, but also likely due to the high perceived costs of maintaining multiple simultaneous contexts.

To summarize, we observed that developers organize episodes in different patterns. These patterns have specific characteristics that enable specific kinds of subgoals. In our study, we found five distinct patterns used by real-world developers. These developers structured their individual episodes into *Sequential*, *Grounded*, *Concurrent*, *Alternating*, and *Recursive* patterns for a variety of subgoals.

C. Maintaining Context across Episodes

RQ3*: How do developers maintain context across episodes?*

The five distinct patterns observed in section IV-B enable different types of subgoals. Each pattern was associated with unique artifacts and interaction with these artifacts. These interaction patterns contribute towards the working context of the developer. A working context is comprised of episodes, which are formed as developers gather information and interact with specific artifacts.

To understand how developers manage working context throughout their goal, we need to understand how developers interact with artifacts and how information flows across episodes. Developers likely gain relevant information through their interactions with artifacts; creating unique information flows for each pattern. To confirm our understanding of information flow across working contexts, we asked validation survey participants to examine several different scenarios and identify the relevant information that contributes to individual actions, episodes, and subgoals. The survey can be accessed from our companion site.

Figure 3 represents responses to the five survey questions related to information relevancy and information flow for specific development patterns. We asked one question per pattern, using neutral language that avoids biasing responses towards any particular pattern. *Information elements* are shown as colored curved lines; The vertical heights correspond to the number of participants that perceived that element to be relevant to a particular action. *Actions* are shown in time sequence order along the horizontal axis; denoted as a_1 through

a_8. Actions are divided into *episodes* and the borders between episodes are denoted by grey vertical lines.

1) ***Sequential Pattern****:* When developers structure episodes in the *Sequential* pattern, consecutive episodes share at least one artifact. These episodes also involve artifacts that are unique to an individual episode.

For example, during his session, P3 decomposed a goal of creating a query interface into three sequential episodes; each towards a different subgoal. Across these three episodes, P3 interacted with the `query_interface` file. All other artifacts were used in exactly one episode. Such observations suggest that participants maintain related information across sequential episodes.

The *Sequential* pattern (see Figure 3a) shows a gradual shift in the relevancy of information i.e. information flows across episodes. Examining the figure further, we see that participants perceived the information from *element 4* (green line) to be relevant across all four episodes. Other than *element 4*, every episode had unique information elements perceived to be relevant to that episode. The peaks of *element 1* (red line), *element 2* (purple line), *element 3* (blue line) and *element 5* (orange line) all show only one rise in relevancy, indicating that these had fairly localized relevancy that did not span across the episodes.

Thus, in the *Sequential pattern, the working context overlaps between subsequent episodes*. Since each subsequent episode is towards a new subgoal, some information and artifacts lose relevancy and are no longer part of the working context. However, there are also cases where some information remains in the working context over a longer period (e.g. *element 4* in Figure 3a).

2) ***Grounding Pattern****:* When developers work through many episodes, towards multiple subgoals, they require grounding in order to reorient themselves to the larger goal. The *Grounding* pattern occurs when developers interact with a different set of artifacts than what is required for their current subgoal.

Figure 3b shows information that participants found relevant in three episodes that adhere to the *Grounding* pattern. The first (a_1 and a_2) and last (a_6–a_8) episodes are instances of "grounding". The second episode (a_3–a_5) denotes an implementation which is then evaluated in the third episode.

We found that, in this case, information is relevant to individual episodes. The information about the implementation from the second episode (a_3–a_5) is no longer relevant once "grounded", as shown in the Figure 3b.

In conclusion, in a *Grounding pattern a working context exists temporarily. Once "grounded", the elements in the working context are no longer relevant*.

3) ***Concurrent Pattern****:* Instances of the *Concurrent* pattern occur when developers switch between subgoals before completing processes intended to address earlier subgoal(s). The *Concurrent* pattern involves two consecutive episodes that do not share artifacts (i.e. developers work with two distinct sets of artifacts).

In the *Concurrent* pattern example described in Section IV-B3, P5 triggered a build (Subgoal 1), and before the build could complete, he began examining the code structure of a particular query command (Subgoal 2). In this example, the artifacts required for each subgoal are distinct and thus can be "concurrently" managed between episodes.

The *Concurrent* pattern indicates that information from one subgoal is irrelevant to the other subgoal. To verify this observation, we asked participants to identify relevant information for actions taken across two concurrent subgoals. The example for the *Concurrent* pattern provided in the validation survey had four episodes across two subgoals (shown in Figure 3c). The first (a_1) and third (a_3–a_5) episodes share the same subgoal, while the second (a_2) and fourth (a_6 and a_7) episodes share a different subgoal.

Figure 3c shows *element 1* (red line) and *element 2* (purple line) to be relevant for the episodes sharing the first subgoal. Whereas information for *element 3* (blue line), *element 4* (green line), and *element 5* (orange line) was relevant for the episodes sharing the second subgoal. The separation between these distinct sets of information, arising from their particular subgoals, provides evidence that information flows between episodes with shared subgoals, and not across episodes with separate subgoals.

From our observations, *two separate working contexts exist when a developer concurrently works on two distinct subgoals*.

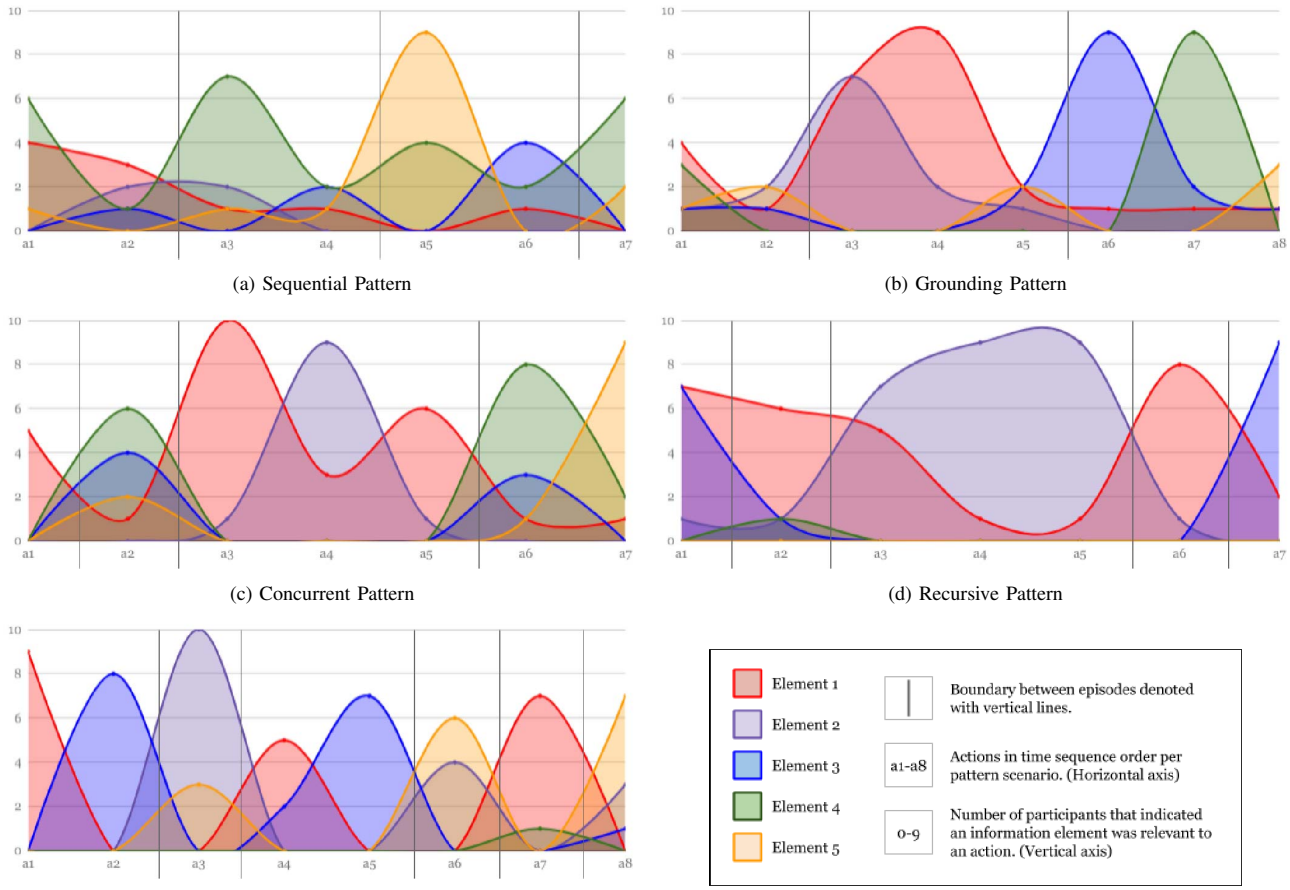
4) ***Recursive Pattern****:* Developers decompose their subgoals recursively into other subgoals. In the *Recursive* pattern, each subgoal is associated with the next subgoal and, thus, a subgoal is likely to tangentially share artifacts across these subgoals.

For example, P3 structured his task into three recursive subgoals. However, he interacted with three distinct sets of artifacts when working towards these subgoals.

Figure 3d shows the five episodes structured into three recursive subgoals for a scenario with a *Recursive* pattern. The first (a_1) and the last (a_7) episodes share Subgoal 1, the second (a_2) and fourth (a_4 and a_5) episodes share Subgoal 2, and the third (a_3 and a_4) episode was directed towards Subgoal 3. Participants indicated that information *element 3* (blue line) to be highly relevant in Subgoal 1, *element 1* (red line) to be relevant for Subgoal 2, and *element 3* (blue line) to be the most relevant for Subgoal 3.

The figure indicates that information flows symmetrically around the lowest level of recursive subgoals. In Figure 3d the axis of symmetry resides at the a_4 action, with the first two episodes (comprising a_1 and a_2) and the last two episodes (comprising a_6 and a_7) residing within their respective halves of the recursive scenario. Additionally, we see that information relevancy mirrors across the axis of symmetry with *element 1* (red line) and *element 3* (blue line) peaking in both halves, and *element 2* (purple line) peaking around the axis.

Based on survey responses, we can conclude that for *Recursive subgoals, developers manage distinct working contexts for each subgoal*.



(a) Sequential Pattern

(b) Grounding Pattern

(c) Concurrent Pattern

(d) Recursive Pattern

(e) Alternating Pattern

Fig. 3: Information Relevancy and Flow for Episode Patterns

*5) **Alternating Pattern**:* In this pattern, developers switch between two subgoals of alternate solutions. Alternating subgoals have two distinct sets of artifacts, which combine together as time moves forward. This suggests that, as developers progress with comparing alternate solutions, information gained from episodes targeting one subgoal can be used in future episodes targeting another subgoal.

For example, P5 tried two alternate solutions to debug a missing dependency error. For the first few episodes, P5 interacted with separate artifacts for each subgoal. However, after the fourth episode, P5 used a combined set of all artifacts when working towards both subgoals.

Figure 3e shows the information our participants perceived to be relevant across these five episodes that represent the *Alternating* pattern. For the first (a_1–a_2) and third (a_4–a_5) episodes, information *element 1* (red line) and *element 3* (blue line) were considered more relevant. For the second (a_3) and fourth (a_6) episodes, information *element 2* (purple line) and *element 5* (orange line) were considered relevant.

In the last episode (a_8), participants indicated that a combination of information elements (previously used separately) were now relevant together. This confirms our observation that, in an *Alternating* pattern, information from episodes towards one subgoal will eventually be used towards another subgoal. Thus, in an *Alternating pattern, developers start with two separate contexts that eventually combine into a single context.*

V. Related Work

A. Task Management

Perry et al. [21] conducted two empirical studies on the social and organizational processes of developers and found that developers work in two hour chunks, spending most of their time on writing code and having unplanned interactions with colleagues. Perlow [22] conducted a qualitative study of how software engineers optimally use their time at work. Gonzalez and Mark [7] found work fragmentation is a common phenomena. Our observations also show work fragmentation among developers, with 242 total distinct work episodes.

Meyer et al. [6], [23] conducted two separate studies to investigate developers' daily activities and observed that developers spend their time on a wide variety of activities, switching regularly between them, and that they perceive context switches to be generally harmful to productivity. However, O'Conaill and Frohlich [24] and Hudson et al. [25] report that

in many cases interruptions can be beneficial and bring relevant information. We find that developers don't just "switch" context, they maintain context to various degrees across these "switches" based on their current subgoal. Information, from both interactions and interruptions, that developers perceive as important flow across related episodes.

B. Information Needs & Flow

Many researchers [5], [26], [27] found that developers perceive questions about the rationale and intent of code was difficult to answer. The majority of questions developers ask relate to the aggregation of information into and across context. Their findings show that developers have significant information needs, which bolsters our research into understanding how developers manage context in their daily activities.

Sillito et al. [28], [29] conducted two qualitative studies of programmers' information needs when performing change tasks. They found that participants asked lower-level questions as part of answering higher-level questions. They further noted that participants sometimes asked linear questions and other times branched out questions about the same entity. We observed similar patterns in how developers structure their subgoals.

C. Context Management

Kersten and Murphy [11] introduce Mylar, a tool which captures the task context of program elements by monitoring the programmer's activity. Gasparic et al. [8] present a context model that includes thirteen contextual factors (in four main categories: *who*, *what*, *where*, and *when*), captured in various situations to enhance interactions within an IDE.

Fritz et al. [30] introduce a model that capture context through developers' authorship and interaction information. Petcharat and Murphy [31] introduce Spyglass, which suggests tools to aid program navigation based on the context of their work. Sedigheh and Murphy [32] captures context through three factors–discovery patterns, recent command usage, and elapsed time since last activity. In this paper we investigate the effect of developer's intention (of what they want to work on) on how they maintain context. We hypothesize that intention and interaction plays equally important roles when modelling context.

VI. Conclusion and Discussions

Our results show that individuals organize their development efforts into a series of episodes, which form different patterns. We found five such patterns: *Sequential*, *Grounding*, *Concurrent*, *Recursive*, and *Alternating*.

Studying context at a smaller granularity (the subgoal level) is important as development efforts typically occurred at the episode level. Furthermore, work thus far typically addresses *how* and *when* developers perform context switches when working across tasks, and their associated cognitive loads [8]. We are the first to observe how context (and portions thereof) is maintained when developers move from one episode to another.

*1) **Implications for Researchers**:* We found that patterns in episodes are associated with the type of development task. For example, debugging tasks were "fast and furious", involved shorter episodes and more switching across subgoals. In contrast, developers were much more deliberate when implementing or refactoring, resulting in longer episodes and less frequent switches between subgoals.

Further research is needed to understand the role of the environment or programming language in task decomposition. While this was not our focus, our survey alludes to this: P1 (working in Java) stated that he "often" performs recursive tasks and ranked *Recursive* as his most frequent pattern.

Task decomposition may also depend on individual differences in problem solving styles [33]. For example, tinkerers who typically operate in small increments are likely to leverage the *Grounding* pattern, whereas planners are more likely to leverage the *Sequential* pattern where they comprehensively process all information needed to solve a task and then decompose it methodically into smaltler, organized subgoals. Further studies will allow us to better understand these differences and design tools that are inclusive to all problem solving styles.

We used the participants' verbalizations to identify when subgoals changed. While this worked well for our qualitative analysis, an automated approach that identifies such boundaries will help in larger studies as well as building developer tools. We plan to experiment with machine learning and natural language processing techniques to automatically identify subgoal boundaries.

*2) **Implications for Tool Builders**:* Modern recommendation tools [9]–[11] typically leverage the relationships between artifacts to recommend other relevant artifacts. Our results indicate that episode patterns impact which artifacts are considered relevant for the current subgoal. For instance, in the *Sequential* pattern, an artifact was consistently used across all episodes. In contrast, in the *Recursive* pattern, artifacts lost and then gained relevance when developers switched between subgoals. Leveraging this correspondence between patterns and artifact relevancy can help improve context-aware artifact recommendations.

Another area of tool improvement is interruption management. Development of tools like FlowLight [34] operationalizes the notion of interruptibility of a developer by using a physical indicator to signal when they are busy. However, such tools either encode interruptibility as a function of time or some physical aspect of developer interaction such as typing speed. In contrast, the end of an episode or a subgoal is likely a better indicator of interruptibility. This nuanced understanding of episode patterns and subgoal structures provide further opportunities to improve interruption notification.

Acknowledgement

We thank our participants for their time, André van der Hoek for his timely suggestions, and the reviewers for their feedback. We also thank Phase Change for their cooperation. This work was funded by NSF 1559657 and 1560526.

REFERENCES

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[2] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, "Foraging among an overabundance of similar variants," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: ACM, 2016, pp. 3509–3521.

[3] Y. Yoon and B. A. Myers, "An exploratory study of backtracking strategies used by developers," in *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*. IEEE Press, 2012, pp. 138–144.

[4] J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, "Productivity factors and programming environments," in *Proceedings of the 7th international conference on Software engineering*. IEEE Press, 1984, pp. 143–152.

[5] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 344–353.

[6] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 19–29.

[7] V. M. González and G. Mark, "Constant, constant, multi-tasking craziness: managing multiple working spheres," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 113–120.

[8] M. Gasparic, G. C. Murphy, and F. Ricci, "A context model for ide-based recommendation systems," *Journal of Systems and Software*, vol. 128, pp. 200–219, 2017.

[9] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 408–418.

[10] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 1–11.

[11] M. Kersten and G. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of the 4th international conference on Aspect-oriented software development*. ACM, 2005, pp. 159–168.

[12] N. Souchon, Q. Limbourg, and J. Vanderdonckt, "Task modelling in multiple contexts of use," in *International Workshop on Design, Specification, and Verification of Interactive Systems*. Springer, 2002, pp. 59–73.

[13] D. Kamma and P. Jalote, "High productivity programmers use effective task processes in unit-testing," in *APSEC*. IEEE Computer Society, 2015, pp. 32–39.

[14] J. Preissle and L. Grant, "Fieldwork traditions: Ethnography and participant observation," *Foundations for research: Methods of inquiry in education and the social sciences*, pp. 161–180, 2004.

[15] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 285–311.

[16] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.

[17] D. R. Garrison, M. Cleveland-Innes, M. Koole, and J. Kappelman, "Revisiting methodological issues in transcript analysis: Negotiated coding and reliability," *The Internet and Higher Education*, vol. 9, no. 1, pp. 1–8, 2006.

[18] A. S. Lee and R. L. Baskerville, "Generalizing generalizability in information systems research," *Info. Sys. Research*, vol. 14, no. 3, pp. 221–243, Sep. 2003.

[19] S. Xu and V. Rajlich, "Dialog-based protocol: an empirical research method for cognitive activities in software engineering," in *2005 International Symposium on Empirical Software Engineering, 2005.*, Nov 2005.

[20] S. J. Tracy, "Qualitative quality: Eight "big-tent" criteria for excellent qualitative research," *Qualitative Inquiry*, vol. 16, no. 10, pp. 837–851, 2010.

[21] D. E. Perry, N. A. Staudenmayer, and L. G. Votta, "People, organizations, and process improvement," *IEEE Software*, vol. 11, no. 4, pp. 36–45, 1994.

[22] L. A. Perlow, "The time famine: Toward a sociology of work time," *Administrative science quarterly*, vol. 44, no. 1, pp. 57–81, 1999.

[23] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, "The work life of developers: Activities, switches and perceived productivity," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1178–1193, 2017.

[24] B. O'Conaill and D. Frohlich, "Timespace in the workplace: Dealing with interruptions," in *Conference companion on Human factors in computing systems*. ACM, 1995, pp. 262–263.

[25] J. M. Hudson, J. Christensen, W. A. Kellogg, and T. Erickson, "I'd be overwhelmed, but it's just one more thing to do: Availability and interruption in research management," in *Proceedings of the SIGCHI Conference on Human factors in computing systems*. ACM, 2002, pp. 97–104.

[26] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 8.

[27] T. Fritz and G. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 175–184.

[28] J. Sillito, G. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 23–34.

[29] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[30] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 385–394.

[31] P. Viriyakattiyaporn and G. C. Murphy, "Improving program navigation with an active help system," in *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2010, pp. 27–41.

[32] S. Zolaktaf and G. C. Murphy, "What to learn next: Recommending commands in a feature-rich environment," in *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on*. IEEE, 2015, pp. 1038–1044.

[33] M. Burnett, S. Stumpf, J. Macbeth, S. Makri, L. Beckwith, I. Kwan, A. Peters, and W. Jernigan, "Gendermag: A method for evaluating software's gender inclusiveness," *Interacting with Computers*, vol. 28, no. 6, pp. 760–787, 2016.

[34] M. Züger, C. Corley, A. N. Meyer, B. Li, T. Fritz, D. Shepherd, V. Augustine, P. Francis, N. Kraft, and W. Snipes, "Reducing interruptions at work: A large-scale field study of flowlight," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 61–72.