A Tale from the Trenches: Cognitive Biases and Software Development

Souti Chattopadhyay[1], Nicholas Nelson[1], Audrey Au[1], Natalia Morales[1], Christopher Sanchez[1], Rahul Pandita[2], and Anita Sarma[1]
Oregon State University, Corvallis, OR[1], Phase Change Software, Golden, CO[2]
USA
{chattops,nelsonni,auau,moralnat,christopher.sanchez,anita.sarma}@oregonstate.edu
rpandita@phasechange.ai

ABSTRACT

Cognitive biases are hard-wired behaviors that influence developer actions and can set them on an incorrect course of action, necessitating backtracking. While researchers have found that cognitive biases occur in development tasks in controlled lab studies, we still don't know how these biases affect developers' everyday behavior. Without such an understanding, development tools and practices remain inadequate. To close this gap, we conducted a 2-part field study to examine the extent to which cognitive biases occur, the consequences of these biases on developer behavior, and the practices and tools that developers use to deal with these biases. About 70% of observed actions that were reversed were associated with at least one cognitive bias. Further, even though developers recognized that biases frequently occur, they routinely are forced to deal with such issues with ad hoc processes and sub-optimal tool support. As one participant (IP12) lamented: *There is no salvation!*

CCS CONCEPTS

• **Human-centered computing** → **Field studies**; **Empirical studies in collaborative and social computing**; • **Applied computing** → **Psychology**.

KEYWORDS

cognitive bias, software development, field study, interviews

ACM Reference Format:
Souti Chattopadhyay[1], Nicholas Nelson[1], Audrey Au[1], Natalia Morales[1], Christopher Sanchez[1], Rahul Pandita[2], and Anita Sarma[1]. 2020. A Tale from the Trenches: Cognitive Biases and Software Development. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380330

1 INTRODUCTION

Cognitive biases—systematic patterns of deviations from optimal reasoning [19, 50, 51]—compromise how humans search, evaluate and remember information. They help us quickly find solutions in a complex world [47]. As a result, they are hard wired in our behavior and occur daily, even in the simplest of situations. A common example is confirmation bias–the tendency to pay more attention to information that agrees with our preconceptions. For example, say we are visiting Korea for the first time, and have been led to believe that the food is very spicy. When we first encounter kimchi, we take a taste and say *"Korean food really is spicy!"*, failing to remember all the non-spicy food we have already eaten. In doing so, we confirm our preconceptions, even in the face of contrary evidence.

Cognitive biases are a result of numerous factors and depend on how individuals–including software developers–think. Some biases occur due to our limited cognitive capacity (e.g. Availability bias may cause developers to choose solutions based on what examples that are readily available in memory), while some are a by-product of an individual's development styles (e.g. Hyperbolic discounting, some developers tend to choose a solution with smaller and more recent rewards as opposed to waiting for larger rewards later on [34]).

While these cognitive biases can result in correct solutions, they can also lead to negative outcomes. For example, in the case of hyperbolic discounting, the small solution that a developer initially chooses might work temporarily but, later, if her chosen (small) solution was not optimal, she could be forced to rewrite all the functionality.

Past works have investigated the harmful effects of specific cognitive biases on software aspects such as, defect density [3], defect proneness [48], requirements specification [18], originality of design [53], and feature design [7]. In fact, Mohanani et al. [34] surveyed 65 primary studies of cognitive biases in Software Engineering and identified 37 cognitive biases in literature.

These studies, however, had been conducted in controlled situations, such as lab studies with student participants, simplified study tasks, and relying upon structured interviews and questionnaire responses. Thus, there exists a gap in our understanding of *how cognitive biases manifest in the real world.* For example, how often and what kinds of cognitive biases occur in software development? How do these cognitive biases affect developers' actions and solutions, and how do developers attempt to mitigate these cognitive biases? Without the answer to such questions, it is nearly impossible to develop effective tools and strategies to help developers avoid the pitfalls of cognitive bias.

To gain insights into cognitive biases and their roles in practice, our first research question (*RQ1: How do cognitive biases affect*

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380330

developer actions?) explores how often cognitive biases occur in real-world development sessions. In addition to the frequency of occurrence, it is also important to understand which cognitive biases developers encounter (*RQ2: What types of cognitive biases do developers frequently experience?*) And based on these different kinds cognitive biases, then *RQ3: What are the consequences of these cognitive biases on developers' problem solving abilities*?

To answer these questions, we conducted a field study of 10 developers while they worked on their routine development tasks, in their work environments. We recorded all activities and interactions with artifacts, resulting in more than 2,000 actions. Our results show that cognitive biases do frequently occur, and can negatively affect development activities.

Given the frequency of cognitive biases, we next investigated how developers guard against or recover from them (*RQ4: What can developers do to overcome these cognitive biases*?), and what tools or practices they use in these efforts. We interviewed 16 developers from three different companies to identify practices and recommend tools that they perceive to be helpful for different types of cognitive biases.

To the best of our knowledge, we are the first to study–*in-situ*–the effects of cognitive biases on development activities. Our results show that cognitive biases frequently occur and negatively impact development activities. We present an initial consideration of tools and practices that developers currently use to guard against or recover from cognitive biases, and have also compiled a set of tool features they seek. Our results are a call to action for researchers to study the cognitive biases—their effects, interplay and remedies—in the wild. Our findings also provide tangible recommendations for tool builders to create solutions specifically catered to reduce the effects of cognitive biases.

2 BACKGROUND

Cognitive biases were first introduced in 1974 by researchers Tversky and Kahneman [51]. Researchers in software engineering have been studying these biases in the domain since 1990 [45]. Mohanani et al. summarizes 65 articles which characterize the current state of studying biases in software engineering [34]. These articles investigate 37 distinct cognitive biases (out of more than 200 previously identified in psychology, sociology, and management research). For example, studies affirm that confirmation bias leads to higher defect rates and more post release defects when testing, availability bias and representativeness bias lead to developers misrepresenting code features, and overconfidence caused insufficient efforts when performing requirements analysis.

Of the 65 papers examined none describe the use of *in-situ* field studies as part of their research methodology. For example, Calikli et al. [4] evaluate the effects of company culture, education, and experience on confirmation bias among software developers and testers through user studies involving interactive and written tests. While lab studies and controlled experiments allow for control of confounding factors, they do sacrifice the richness and spontaneity of naturalistic observations [10]. Calikli et al. [5] also observed developers in hackathons to understand different levels of confirmation bias. While hackathons have higher ecological validity than lab studies, they still do not mimic developers' natural work environment. Our study attempts to extend these lab-based (or studies in non-natural environment) findings to actual development practice via the use of observational studies in a real-world setting.

Importantly, examining biases within software development tasks using empirical methods such as observation studies, requires precise definitions of measurements and related constructs [10]. To this end, we define *goals* as the end state towards which a developer directs her development effort, and *actions* as steps performed by her to reach that goal [6]. Our coding schema and additional description of its usage is expanded in Section 3.

Related Work. Prior studies have examined developer behavior and activities to understand their practices in terms of problem solving [29, 36, 52], tool-usage [12, 22, 35], information search and use [26, 46], and work context [6, 20]. These studies, none of which utilized a field study as a means of collecting data, help us understand developer work practices [31, 32], where they struggle [2, 11], and what information they need [23, 44].

Other studies have also investigated practitioners' perceptions or beliefs. For example, Devanbu et al., [9] explored the relationship between quantitative evidence and practitioner belief, and found that developers often rely on individual experience. Meyer et al. [32] and Pandita et al. [37] investigated how developers solve problems based on their perception of productivity and effort. While these perceptions and beliefs may very well stem from cognitive biases, we do not know which biases are presenting themselves, nor their specific effects. Our work explicitly attempts to identify specific cognitive biases, and understand their effect on developer work.

3 METHODOLOGY

3.1 Field Study

We observed 10 developers *in-situ* for our field study. Our participants were recruited from a US-based software startup (company A) that specializes in the areas of distributed developer tools and services, including: program analysis, user interface (UI) design, infrastructure support, and software R&D. Due to the diversity of focus areas in the startup, participants used a wide variety of programming languages, tools, and working styles.

Table 1 presents demographic information about study participants, including development experience, code editor usage, and preferred programming languages (the median software development experience was 2 years, and the mean was 5 years 9 months).[1]

We observed participants performing their routine development tasks on a typical workday. During the observation session, we asked participants to think-aloud and verbalize their thoughts and interactions. [40]. We recorded their screen, audio, and physical workspace.

The total observation time per session was limited to 60 minutes to prevent participant burnout and respect time restrictions at the startup.

During each session, one researcher was positioned next to the participant, taking notes. In a separate room, not visible to the participant, an additional researcher served as a secondary field-note taker, and monitored the recordings of the participant to ensure

[1]We could not study whether gender had any association with biases as very few participants in our study were women.

Table 1: Study Participant Demographics

Ptc.[i]	Gnd.[ii]	Exp.[iii]	Language(s)[iv]	Editor[v]
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, Clojure, Java	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] Gnd. = Gender [iii] Exp. = Years/months of software development experience [iv] Preferred programming language(s) [v] Editor used in session

consistency. Members of the research team alternated between serving as the primary and secondary note takers.

At the end of each session, participants were asked to complete a brief demographic survey (see results in Table 1) while the two researchers compiled and prepared follow-up questions to clarify during a 15 minute retrospective interview.

3.2 Qualitative Coding

To understand how cognitive biases affect software development, we identified and classified each individual action taken by participants during our observational study using qualitative coding methods.

To code the raw data, we first transcribed all data including: 1) participants' verbalizations from the think aloud recordings, 2) descriptions of actions taken by participants, and 3) the artifacts interacted upon by the participants. Each piece of data, called an instance, contains a quote and description of the participant's actions.

Table 2: Action Codes

Action	Definition
Read	Examining information from artifacts (e.g. code, documentation, terminal output).
Edit	Any change made directly to code or 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.

3.2.1 Action coding. An action is a discrete step performed by a participant to achieve their task. To code the actions in our transcript data, we created five sets of 94 (4.5%, total 22.5% over five sets) random instances from the observations across all participants. Three authors individually coded each action with the codes described in Table 2.

Using Fliess' Kappa to calculate the inter-rater reliability (IRR) measure for three coders [24], we achieved kappa values of 0.82, 0.80, 0.87, 0.88, and 0.87 for the five sets, respectively $p\text{-}value < 0.001$. After achieving the recommended threshold agreement, the three researchers split the remaining dataset, and individually coded the remaining actions.

Participants worked on three types of tasks when taking these actions: implementation (1119 actions), debugging (535 actions) and verification (731 actions). We found this through qualitative coding (IRR of 0.86 across 3-raters; Fleiss kappa). *Implementations* actions refer to adding new feature or functionality (e.g. adding wrapper class), *verification* actions refer to verifying new or modified functionality to check if added code executes (e.g. Compiling, executing code), and debugging actions refer to identifying and eliminating bugs in existing or added code (e.g. after error message, changing query syntax in console).

3.2.2 Bias coding. We examined each action to identify the associated bias categories. These bias categories are explained in Section 4; including discussion on how each category was observed.

To ensure the validity of the coded biases, three authors incrementally coded the entire dataset using *negotiated agreement* [13].

3.3 Complementary Interviews

We next conducted semi-structured interviews with 16 developers to triangulate our findings from the initial field study. This additional data collection was designed to develop a formative understanding of how developers both perceive, and deal with, the observed biases. We used semi-structured interviews instead of surveys to: (1) confirm that participants correctly understood the biases, while allowing them to ask clarifying questions, (2) follow-up on advice/practices that participants suggested to address bias.

We recruited interview participants from three companies to examine debiasing practices across a variety of organization sizes and cultures. First, we observed 11 developers from the original company in our field study (Company A); from our observed 10 participants [see Table1], two employees had left and three others had joined since our field study. Next, we interviewed one developer from another start-up of similar stature (Company B); team sizes similar with those at Company A. Finally, we interviewed four developers from a Fortune 500 company (Company C); a multinational company with large team size. These interviews helped to confirm that the observed biases were not limited to a single company. Table 3 provides demographic information for all of the interview participants.

In the interviews, we defined ten bias categories and provided examples based on instances observed in our field study (see Section 4 for definitions). Using these generalized examples, we asked two questions for each specific bias: (1) "On a scale from 1 (low) to 5 (high), how often do you think developers act under this bias?" and (2) "What standard practices, guidelines, or tools would help to avoid this bias?"

Interview responses were categorized by two authors using Pattern Coding [33]—the process of grouping categories into smaller thematic sets. In the first round of qualitative coding[41], we identified 29 development practices (e.g. brainstorming, referencing) described in response to the second interview question. In the second round, these practices were abstracted into five categories that link specific biases with practices that directly address them (see

Table 3: Interview Participant Demographics

Pt[i]	Gn[ii]	C[iii]	Exp[iv]	Role[v]	Pt	Gn	C	Exp	Role
IP1	M	A	23y 0m	Dev	IP9	M	A	8y 0m	Dev
IP2	M	A	2y 11m	Dev	IP10	M	A	2y 0m	Dev
IP3	M	A	2y 10m	Dev	IP11	M	A	1y 9m	Dev
IP4	M	A	8y 3m	Dev	IP12	M	B	1y 0m	Dev
IP5	M	A	3y 0m	Dev	IP13	M	C	5y 0m	Dev
IP6	F	A	19y 8m	QA	IP14	F	C	2y 0m	Dev
IP7	M	A	6y 0m	Dev	IP15	M	C	2y 0m	Dev
IP8	M	A	19y 0m	Dev	IP16	M	C	5y 0m	Dev

[i] Pt. = Participant [ii] Gn. = Gender [iii] C. = Company [iv] Exp. = Years/months of software development experience [v] Job position in the company

Table 6 for details). We also aggregated the tools participants found helpful for specific practices into these categories.

4 BIAS CATEGORIZATIONS

A description of individual biases can be found on our supplemental site[2]. This site provides annonymized supplemental artifacts used for data analysis. We cannot release raw data due to participant privacy concerns.

Table 4 lists the 28 biases observed (out of the 37 reported by Mohanani et al. [34]); the remaining 9 were likely not observable because of our study design–an hour long observational study with think aloud protocol. Additional details are available on the supplemental website[2].

4.1 Bias Categories

We grouped the *28 observed biases into 10 categories* based on their effect on developer's behavior. These categories were created through the process of negotiated agreement. Two authors individually categorized the 28 biases by combining: (1) the definitions of cognitive biases in the context of software engineering (per Mohanani et. al. [34]), (2) the definitions of biases in cognitive science literature [50, 51], and (3) the observed effects of biases on participants' development behavior (through direct observation and participants' verbalizations). In the first round, the authors agreed on the categorization of 24 out of 28 biases (85.7% agreement), into a set of 11 categories. In the second round, the authors disagreed on one bias categorization (96.4% agreement) and decided to merge the 1st and 11th categories. Table 4 shows the final list of 10 categories (CB1–CB10), and their mapping to individual cognitive biases.

The *Preconceptions* (CB1) category refers to the tendency to select actions based on preconceived mental models for the task at hand. Biases within this category cause developers to discount the degree of solution space exploration required to take action.

Ownership (CB2) occurs when developers give undue weight to artifacts that they themselves create or already posses, thereby reducing the potential for other options to be objectively evaluated. Preference for one's own artifacts prevents developers from exploring the solution space completely.

The *Fixation* (CB3) category refers to anchoring problem solving efforts on initial assumptions, and not modifying said anchor sufficiently in light of added information or contradictory evidence. This leads to reduced awareness of task context.

Resort to Default (CB4) occurs when developers choose readily available options based solely on their status as the default, or the tendency to prefer current conditions without regard to applicability or fitness. This causes lost context of the overall task.

The *Optimism* (CB5) category reflects the set of biases that lead to false assumptions and premature conclusions regarding efficiency or correctness of a chosen solution. This is shown when people overtrust their abilities, or when the likelihood of a favorable outcome is over-estimated.

The *Convenience* (CB6) category encompasses the assumption that simple causes exist for every problem, and the predisposition to take the seemingly quicker or more simplistic routes to solution. This reduces the effort developers invest in reasoning and making sense of information.

The *Subconscious action* (CB7) category refers to the offloading of evaluation and sense-making to external resources (such as IDEs or online resources) without regard to the actual merits of such information.

Blissful ignorance (CB8) refers to the assumption that everything is nominal and working, even in the face of information indicating otherwise. Developers don't pay attention to their surroundings.

The *Superficial selection* (CB9) category represents a range of actions and information being unduly valued based on superficial criteria. Developers decide on a solution without thoroughly reasoning through it.

The *Memory bias* (CB10) category affects how developers remember information from a series of alternates, prefer to use the primary or most recent information encountered, or react as a result of information most readily available in the memory.

5 RESULTS

5.1 Presence of Biases in Developer Actions

The field study included 2084 distinct developer actions; of these we classified 953 actions that contained at least one bias category. Thus, approximately half of developer actions (45.72%) were associated with some form of bias. Note, the large number of biased actions (953 out of 2084) in our observation time frame may likely be due to cognitive biases being inherent in decision making actions, which are a key part of software development.

However, not all cognitive biases necessarily result in a negative outcome. Biases can lead to positive effects–participants taking fewer actions than anticipated. However, in a non-controlled environment we cannot differentiate between the "baseline" (no-bias) or "optimized" (positive outcomes of bias) number of actions. Thus, we focused on reversed actions (negative bias).

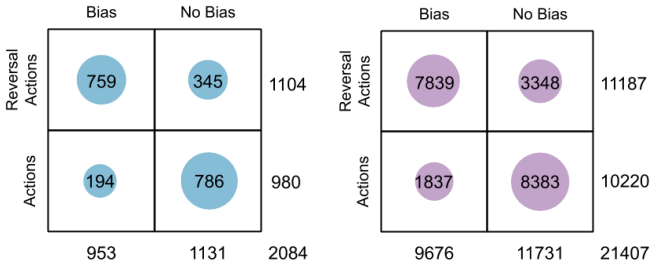
To identify biases that resulted in a negative outcome, we use the notion of `Reversal Actions`. We define `Reversal Actions` as the actions that developers need to undo, redo, or discard at a later time. Reversal actions are thus indicative of non-optimal solution paths.

Figure 1a shows the distribution of developer actions (biased or non-biased), and whether it led to a negative outcome. There were 953 actions with biases, and 1131 without. Similarly, there were 1104 reversal actions and 980 non-reversal actions. Reversal actions were more likely to occur with a bias – 68.75% (759/1104 cases), and

[2] https://epiclab.github.io/ICSE20-CogBias/

Table 4: Cognitive Bias Categories

	Bias Category	Bias(es)	Example
CB1	Preconceptions	Confirmation, Selective perception	P1 continually added hashmaps when other data structures were more suited for data query APIs.
CB2	Ownership	IKEA effect, Endowment effect	P8 decided to reuse her old CSS file instead of the pre-made CSS files from the Bootstrap project.
CB3	Fixation	Anchoring and adjustment, Belief preservation, Semmelweis reflex, Fixation	P9 fixated on changing the function definitions when the environment just needed to be reloaded.
CB4	Resort to Default	Default, Status-quo, Sunk cost	P2 opened a new code file and kept unused template code at the top of the file.
CB5	Optimism	Valence effect, Invincibility, Wishful thinking, Overoptimism, Overconfidence	P4 was proud of his new aggregating map code, but it got an error after it was printed.
CB6	Convenience	Hyperbolic discounting, Time-based bias, Miserly information processing, Representativeness	P2 created simple overly-verbose code that addressed his current needs, but became spaghetti code that slowed future progress.
CB7	Subconscious action	Misleading information, Validity effect	P6 focused on fixing the files listed in error messages instead of the core dependency file causing errors throughout the system.
CB8	Blissful ignorance	Normalcy effect	P10 disregarded all compiler warnings out of habit and failed to notice a new exception detailing the cause of his build failure.
CB9	Superficial selection	Contrast effect, Framing effect, Halo effect	P4 copied and pasted a function from his documentation directly into his syntax without examining it first.
CB10	Memory bias	Primacy and recency, Availability	P1 reused a design pattern that worked well on recent tasks, since he could easily recall the structure of the code.



(a) Distribution of Actions **(b) Distribution of Time (sec)**

Figure 1: Distribution of Presence of Bias and Reversal Actions. Size of circles are proportional to (a) the number of actions or (b) time (in seconds). Each cell presents the actions matching these dimensions. Totals are shown along the bottom and right edges, with overall totals shown in the lower right-hand corners.

biased actions were more likely to be reversed – 79.64% (759/953 cases).

To verify this association, we conducted a chi-square test of independence with a Bonferroni correction (to account for multiple comparisons [42]). To be considered significant, the p-value needs to be ≤ 0.0125 (adjusted $\alpha = 0.05\times(\frac{0.05}{4}) = 1.25e{-}2$) when applying Bonferroni correction to a chi-square test across 4 comparisons (2 categories for whether actions were biased and 2 categories for whether actions were reversed). The chi-square test is significant ($\chi^2[4, N = 2084] = 499.35$, p-value $< 2.2e-16$). Thus, biased actions were highly associated with reversals.

To evaluate the strength of this association we estimated the Cramer's V measure (for 2×2 comparisons) [39]. $V \geq 0.50$ is considered a large effect size when the minimum number of categories in either variable is 2 [43]. The Cramer's V results signify a strong association between the presence of bias and actions that need to be reversed (V= 0.5).

However, if the time spent on reversing actions is not substantial, the number of reversal actions alone might not provide an accurate estimate of the negative outcomes of biases. Therefore, we analyzed the time spent during each action, captured in Figure 1b. Each cell presents the time (in second) spent in each type of action.

In total, developers spent 34.51% (7839/21407) of their time reversing biased actions. When focusing only on the time spent in reversal actions, 70.07% (7839/11187) of these involved biases. Therefore, biased actions lead to significant negative outcomes in terms of lost development time (approximately 25% of their entire working time). A chi-square test of independence supports this hypothesis, showing that the time spent reversing actions is not independent of biased actions–$\chi^2[4, N = 21407] = 5850.2$, p-value $< 2.2e-16$ showing significant results with Bonferroni corrected α, and large effect size with Cramer's $V = 0.5$.

> *Not only are biases frequently present during development (45.72%), but also biased actions are significantly more likely to be reversed later. Furthermore, developers spend a significant amount of time reversing these biased actions.*

5.2 Distribution of Biases

To understand how to reduce the negative effects of biases, our analysis focuses on the reversal actions associated with biases (759 actions in Figure 1(a)). For example, P4 ended up reversing 91 actions (which he spent 699 seconds) because he was fixated (CB3) on his hypothesis that the syntax of the query function was causing an error. In addition to Fixation, our participants incurred 9 other types of biases (Section 4).

Figure 2 shows the distribution of these bias categories—CB3 had the highest instances of reversal actions (428 reversals), and CB9 had the lowest instances (8 reversals). Note, a reversal actions can be a result of multiple bias categories. For example, P4 'read' the documentation of the query function because he was Fixated (CB3) and that function was the first thing they remembered(CB10). In this case, we counted the 'read' action once for both the CB3 and CB10 categories. Since our analysis focuses on the effects of a specific bias category on a developer action, this coding scheme does not impact our analysis.

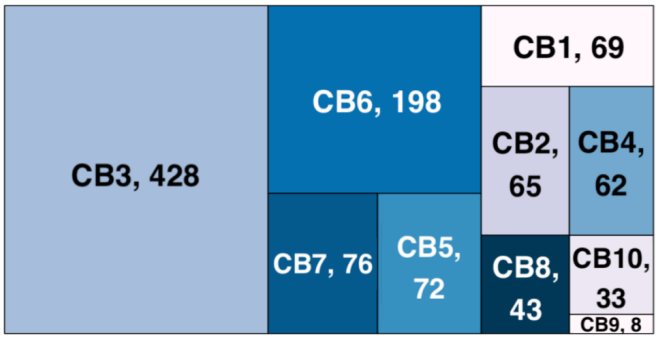


Figure 2: Distribution of Biased Reversal Actions According to Bias Category. Rectangle sizes are proportional to reversal action counts for each bias category; category code counts are shown in each.

Next, we look at the instances of the reversal actions within each bias category to understand when and why these are likely to occur. We discuss these biases in descending order of frequency.

Fixation (CB3) bias was the most frequently occurring bias category; 477 developer actions were associated with this bias, of which 428 were reversed. A recurring scenario was when participants 'fixated' on a specific solution—rejecting warnings and errors that contradicted their beliefs—as they continued to pursue their solution. For example, P5 got connection errors when trying to compile code. He became fixated on the connection issue; not only resolving the issue but also improving the infrastructure to avoid similar issues in the future. Because of this fixation, P5 lost sight of his original task. Even after working on this improvement for 6 minutes and 5 seconds, his code still failed to compile which then made him switch to his original task. The longer a participant fixated on an (incorrect) solution, more changes required reverting.

Convenience (CB6) bias was the second most frequent bias; 245 developer actions were associated with this bias, of which 198 were reversed. This bias occurs when individuals resort to the easiest, or most convenient solution, even if the solution is not optimal. For example, while using an array to simulate a graph, P1 implemented a method to convert the first array element into the starting node. However, P1 failed to address the case where the first element was empty. Only after several failing tests did he realize that his implementation was based on an over-simplified hypothesis, and he had to re-implement the method. Such rework was a common occurrence with convenience bias, as participants with this bias performed limited explorations which resulted in incomplete or incorrect solutions.

Subconscious Actions (CB7) All 76 actions associated with this bias were reversed. This bias frequently occurred when participants blindly trusted their tools or IDE. For example, P10 executed her test case but didn't observe the status of the tests and started adding code to the repository. She believed that the the tool would notify the status of the test via an email when the test case execution terminated. However, when she didn't receive any email after several minutes, she went back to the test cases. She realized *"Testing finished, but it didn't send an email so it's stuck somewhere on these results"*. Eventually, she found out she needed to update the data for the tests to run successfully.

Optimism (CB5) Optimistic actions were reversed for 72 out of 73 times. This was common when initial verification results looked promising. Participants were most optimistic about their changes once they received initial confirmation. Following which, they made larger changes without thoroughly verifying their solution. We observed this when P6 was undertaking a major refactoring within a particular module and relocating files in the process. After P6 completed the refactoring and found no warnings from the IDE, he convinced himself that the task was complete. Eventually, run-time errors showed that associated paths still needed to be updated.

Preconception (CB1) bias was associated with 69 reversal actions out of a total 115 actions. For example, P4 tried to use a new command to check for specific elements while debugging a newly created data structure. When he received a syntax error instead, he modified his command eight times based on documentation suggestions and what he thought might work. Eventually, P4 concluded that the new data structure did not allow membership checking. We also observed participants reversing preconceived actions when using tools (such as the debugger and developer tools) to verify portions of code they suspected were functioning improperly. For example, P8 used the developer tools in her browser to verify that her changes were appropriately reflected on the web interface; verifying her preconceptions about the key elements to be changed before committing to those changes.

Ownership (CB2) bias actions were reversed for 65 out of 77 total actions. Participants acting under this bias sometimes believed that the objects they've created were superior to other options, preventing them from considering other options that may have worked better. They prioritized familiarity saying, *"Lemme go ahead and use what (already) did . . . "* [P8]. We observed this bias when P2 was attempting to create his own `namespace` in Java before considering other options. He eventually stopped and tried a different solution, as the initial attempt was time consuming and didn't work.

All 62 actions associated with **Resort to Default (CB4)** bias were reversed. Participants expected less errors with existing code, code they recently pulled from the repository. When errors were present, participants took longer than average to locate and fix them. For example, when P10 used an existing data file to test new visualization code, the tests failed with a "response mismatch" error. The code for extracting data from the file was copied and used without modification, since the defaults were assumed to be correctly configured. However, upon further inspection, P10 realized that the extraction code required updates to correctly interface with the new visualization code.

Blissful Ignorance (CB8) bias actions were reversed for 43 out of 44 instances. This is because developers operated under the

assumption that everything is working, despite the information indicating that the opposite is true. For example, we observed P9 looking for something to verify if a list contains a `vect`. He repeatedly types in his keyword, then tries a different keyword, then changes his syntax, then hypothesized that something wasn't working with his logic, finally realizing that his prompt in the environment wasn't working. When he reloaded the environment, he needed to go back to when he started his started his experimentation to see which of his solutions worked.

Memory (CB10) bias actions needed to be reversed 33 times out of 43 instances. Participants often would try a solution, even though they only partially remembered the solution and its outcome without reasoning about whether the solution fits the task at hand. For example, P2 tries to implement a membership checking function because it's the first one he remembers. He ends up spending about 5 minutes searching for the function he thinks he remembers, even though he could have found a more appropriate function by a simple online search. Sometimes, participants spent time on artifacts they perceived were relevant based on a biased memory. For example, P10 spent 6 minutes trying to locate and remove an 'outdated' json file she recalled. She searched for the file from the explorer, terminal as well as remote repository. Eventually she said, *"So that one[file] wasn't there to begin with."*

Finally, **Superficial Selection (CB9)** bias actions needed to be reversed for all 8 observed instances . This bias caused participants to quickly decide on a solution based on its aesthetic or appearance. When P4 referred to the language documentation to find a function to concatenate elements of two different types, he chose a function that *"sound(ed) right"* to him, but was ultimately incorrect.

Bias categories affected developers uniquely, causing reversal actions. Fixation(CB3) was the most common and were associated with the most number of reversals, followed by convenience(CB6) and subconscious action(CB7) biases. Memory(CB10) and superficial selection(CB9) biases had the least number of reversals.

5.3 Consequences of Biases

To identify the consequences of these biases on development, we investigated the effects of bias categories on participants' decision making and problem solving.

Two authors categorized the effects of the bias categories into four consequence groups by studying the instances of the biases through negotiated agreement. During the first round of negotiation, the authors only had three groups, and disagreed on the classification of one category (CB8) (90% agreement). During the second round, the authors resolved the disagreement with CB8, and identified a fourth group (attention) which was not properly addressed by the first three groups. Table 5 shows the consequences of biases on development and the associated categories.

The authors identified the effect of each bias category on four orthogonal problem solving activities in programming: gathering information [44], making sense of the information [16, 39], and maintaining information (context) that is relevant to particular tasks and goals [6, 14, 15] and maintaining and focusing attention in the necessary places [38]. Each consequence, and an example from our observation, is described below.

Table 5: Consequences of Biases

Consequence	Bias Categories (CB)
Inadequate Exploration	1, 2, 4, 5, 6, 10
Reduced Sense-making	5, 6, 7, 8, 9
Preserving Context	3, 5, 8
Misplaced Attention	3, 4, 8, 9

Inadequate Exploration– Exploring or foraging different pieces of information [38, 44] and evaluation of alternate solutions [17, 21, 30] forms a key part of development. Cognitive biases sometimes inhibited participants from investing in proper exploration.

Reduced explorations often led to participants creating suboptimal solutions, which they had to redo. For example, P4 needed a subset of data from a hashmap which required him to query the hashmap. As he was not familiar with the query interface and did not how to construct the query, he decided that an easy-fix (CB6) was to manually collect the data.

[14:26] *"Easiest thing to do would be to collect all input statements and instead of using the query, do this myself."*

P4 then began implementing this functionality under the preconception (CB1) than manual data collection is easy. However, after trying this for the next 18 minutes, he realized that the implementation was far more difficult than what he had expected, at which point he decided to learn how to query a hashmap.

Reduced Sense-making– Sense-making is the process of cognitively engaging with information to construct a relevant mental model which can then be used to understand a given situation [8, 16, 39]. Actions that indicate synthesis of knowledge through reasoning are considered examples of sense-making. We identify reduced sense-making through participants' verbalization indicating previous actions (and assumptions) were incorrect.

For example, P10 was testing modifications to data pipelines (used to aggregate and monitor data) and found that her tests failed after she added new input data files to the pipeline, and said "…we're getting this fail response. Which shouldn't be …". She subconsciously (CB7) followed the error location suggested in the message without reasoning about the error. She spent 5 minutes trying to debug her data files but failed to locate the error. Eventually, she found that she was using older input files which caused the error; her tests worked after she updated these files.

[13:25] *"…So that one [file] wasn't there to begin with …"*

Context Loss–When navigating and making sense of different sets of information, developers must retain a mental model of the problem space and relevant information to complete a task [6, 25].

A reduction in context can be seen when participants repeatedly backtrack or verbalize confusion regarding the current task or goal (i.e. losing track of their current actions). Biases can inhibit the ability of developers to create and maintain context, either through a lack of awareness of changes to relevant information or reduced ability to recall elements from the context.

This is shown when P3 fixates(CB3) on trying to solve an error and gets sidetracked, thus losing the larger context of their task. We also observed that when participants were optimistic (CB8) about an implementation, they would suspend the related context and

move on with their task. These participants struggled to recall the context at a later time when their implementation failed.

Misplaced Attention– Attention is a critical element of our cognitive system, and affects what information developers perceive as relevant, how developers interpret error messages, and what solutions they decide to pursue. Biases can cause developers to misplace their attention on peripheral or inappropriate information, causing them to spend time working on issues that are irrelevant to the current task.

When P4 tried to debug his query function which was returning `nil`, he thought the problem could be an incorrect query syntax.

[26:28] *"This is the API for Clojure and I'm looking for something that tells me how to check if a list contains a* `vect`*."*

He became so focused on changing his syntax (CB3) that he didn't notice the syntax highlighting was no longer working and his environment had failed (CB8). After trying five different tweaks to the query function on the inactive environment, he noticed that the syntax highlighting was not working. He killed and refreshed his environment, and had to recall and try all the five tweaks again.

Biases affect multiple aspects of problem solving during development. Specifically, biases affect how adequately developers explore the solution space, how thoroughly they engage in sensemaking, how effectively they retain context, and how efficiently they invest their attention.

5.4 Dealing with the Consequences of Biases

To further bolster our finding that biases occur frequently in practice, we interviewed 16 developers asking them: "how often do you think developers act under [bias category]?" (see Section 3.3).

Figure 3 shows the perceived frequency of occurrences for each bias; ranging from "Almost never" to "Always". The frequencies are indicated by hues of red. Dark red bars denote high frequencies (Often, Always), with their percentages reported far right. (E.g., 81% of interviewees considered CB10 to be frequently occurring). Light red bars denote low frequencies (Almost Never, Rarely), with percentages reported far left (e.g., 6% found CB10 to be infrequent). Grey bars in the center reflect the frequency of "sometimes" responses, which is considered neutral within subsequent analysis.

Overall, interviewees felt that biases occur frequently in software development (Figure 3), matching our observations. For example, when talking about Convenience bias (CB6), IP12 said:

[32:12] *"It happens all the time! . . . It's the story behind why technical debt happens! Three months and then you go and ask why on earth is this failing? And when you look back and somebody overwrote something because it was easier. And it screwed up everything!"*

Memory (CB10), Convenience (CB6), and Preconception (CB1) bias were ranked highest in perceived frequency. These ratings are, at least, partial confirmation of our empirical findings, as Convenience(CB6), Fixation(CB3), and Preconception(CB1) were likewise in the top 5 most observed biases (Figure 3 and Figure 2). However, Memory bias(CB10) and Subconscious Action(CB7), seem to diverge in terms of the actual frequency of demonstration, and the perceived frequency of.

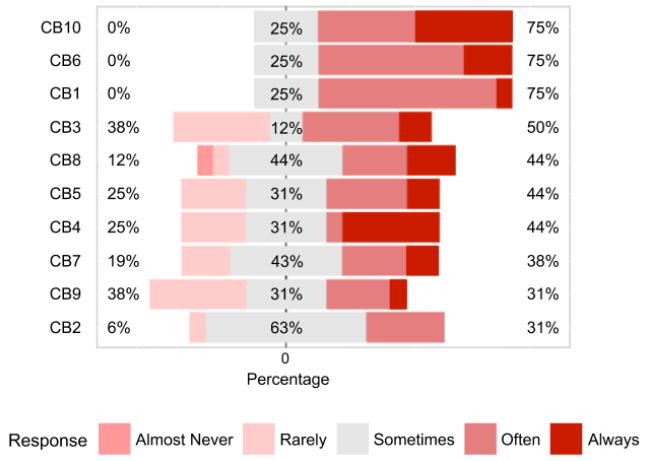


Figure 3: Perceived Frequency of Biases from Interviews ranging from 'Almost Never' to 'Always'. Bias categories are ordered in descending order from most to least frequent (shown as percentages).

Developers perceive Memory bias (CB10) to occur more frequently than what we observed, whereas, Subconscious Action(CB7) was more frequent in our observation. These differences could be attributed to the unconscious nature of these biases. For example, it is difficult for developers to consciously distinguish what or how they remembered a given piece of information, especially in hindsight. Similarly, developers might not be fully aware of how often they subconsciously act based on environmental cues. Probing this disparity between observation and perception could prove a very interesting line of future work.

5.4.1 Practices that help: Although current development practices and tools are not designed to avoid cognitive biases, developers might still be using them to do so. Therefore, during our interviews, we asked participants to identify practices and tools that could help them (or their coworkers) avoid or recover from biases. There were 246 unique suggestions from these interviews.

Analysis: Two authors categorized the suggestions using Pattern Coding [33]—the process of grouping categories into smaller sets of themes. Three themes emerged—Development Practices, Who performs these practices, and When. In the first cycle of qualitative coding[41], 29 categories emerged (e.g. brainstorming, referencing). In a second cycle, these practices were reorganized into 5 categories.

Table 6 displays these categories and themes, along with the biases that these practices can help. The last column lists the tools that participants found useful to help with these practices. The 'Categories' column indicates the category of the practice, and the 'Subcategories' column describes helpful practices within the category. The 'Who' column indicates whether the team (T) needs engage in the practice or the individual (I) can do it themselves. The 'When' column specifies whether the practice needs to be done Before (B), During (D), or After (A) a task. Lastly, the 'Biases' column indicates the biases each practice can help with. We next describe these development practices in detail:

Stepping Back: Taking a break from one's own development pattern can help developers become aware of beneficial practices (like clean code), which can avoid biases like Preconception(CB1)

Table 6: Helpful Practices

Categories	Subcategories	Time	Who	Biases	Tools
Stepping Back	**Incentivized Training:** discussion of clean code benefits, long term goals	B/D	T/I	2,6,7	NA
	Non-Code Days: documentation days, test fest, familiarize with concepts	D	T	1,6,10	
	Meaningful Configurations, updating configurations, meaningful defaults	B	T	4,10	
Different Perspectives	**Confer With Developers:** pair programming, collaborative decision brainstorming (code/design/tool), verify global changes, designated tool guy	B/D	T	1,2,3,6,9,10	Slack, Hipchat
	Open Communication: encourage communication, communicate early with teams like QA. Promote focus on functionality and need	D	T	2,5,6,7	
Systematic Approach	**Systematic Exploration:** prior research on tools, compare & contrast solutions, problem decomposition	B	I	1,2,3,4,5, 6,7,8,9	Dev Tools, Sonarlint
	Big Picture in Mind: re usability of code, backwards compatibility	D	I	1,2,6,10	
	Consistent Early Feedback: reviews (design, expert, peer), sprint meetings	D/A	T	1,2,3,4,5,6	
RTFM	**Reference Doc.:** req/API/design doc, code comments, online sources	B	T	1,4,5	IDE Suggestion
	Journal Options/Alternatives: playbook, team diary	B	T/I	1,2,10	
	Meaningful & Relevant Specifications: standard specs, severity/relevant levels for warnings, protocol for resolution of warnings, descriptive errors	B/D	T	2,4,7,8	
Processes	**SE Concepts:** agile, code review, shared artifacts, reduced ownership, design first, UML diagrams, user story, TDD, BDD, constant debugging, data flows	B/D/A	T/I	1,3,4,5,6, 7,8,9,10	ZenHub, Gerrit, Debugger, IDE, JIRA, JaCoCo
	Standardization: corporate/coding/package standards, right arch. & microservices, clean code, performance test, impact analysis	A	T	1,2,5,6,7,8, 9,10	
	Problem Solving Strategies: divergence & convergence thinking, defensive programming, negative hypothesis testing, timebox, note todos in code	B/D/A	I	1,3,5,6,7,8,9	

and Memory(CB10). For example, IP13 described *Documentation days* and *Test Fests*, as:

[20:01] *"…documentation days are where we say, 'Today, we're not going to be writing code. We're going to focus on checking the documentation, updating documentation …' You might run into some of the new methods …and then you are more bound to use them next time."*

Similarly, learning through focused and incentivized training can help ingrain "good" practices that will help developers avoid biases like Convenience(CB6). For example, IP14 mentioned how "clean code" workshops were:

[20:06] *"really instilled in all of us–oh, it really matters to build the highest quality code!"*

Different Perspectives: Appreciating a different perspective, coupled with associated relevant feedback, can help avoid biases such as Preconception(CB1), Fixation(CB3), and Superficial Selection(CB9). Being exposed to different methods can help break developers out of cognitive 'boot loops' by forcing them to reconsider, evaluate, and justify any subsequent action. For example, pair programming can help with Superficial Selection(CB9) as the navigator can point out any errors in reasoning when programming.

Systematic Approach: To avoid falling victim to biases or other errors, individuals should systematically approach the problem space and explore available solutions and tools. Such systematic review of different task parameters can help in avoiding biases such as, Preconception (CB1), Memory (CB10), and Fixation (CB3) as developers will be both better aware of potential pitfalls, and also can consider alternate solutions ahead of time.

[14:49] *"…[when choosing a tool] one of our criteria was [researching] how well is it documented? And I think it [good documentation] is very important."*

In addition to alternate solutions, systematic exploration helps developers keep the 'big picture' in mind. In other words, it forces developers to more explicitly appreciate and acknowledge the larger goal, hopefully minimizing the likelihood that they will be distracted when *in situ*. This can prevent biases such as Ownership(CB2), by promoting the use of existing relevant code (that does not necessarily just belong to a single developer), which helps keep the larger code base backwards compatible.

RTFM: Consulting documentation before starting a task can avoid biases such as, Preconception (CB1), Memory (CB10), and Ownership (CB2), as developers can become aware of the multiple ways to problem solve and the pitfalls of each solution. For example, *Team diaries* and *playbooks* are journals where developers record guidelines for libraries and packages that specify how to use any code artifacts and avoid pitfalls.

Standardized, descriptive documentation of how to handle errors (or warnings) along with their severity levels can also help overcome biases such as, Blissful ignorance (CB8) and Optimism (CB5) by helping developers locate faults more quickly.

Processes: Good software engineering practices like designing and testing early and frequently, agile software development, etc. can help avoid biases in all categories to some extent. Developers can avoid biases such as, Ownership (CB2), and Resort to Default (CB4) through coding standards and the use of standard libraries. This also helps developers to locate appropriate code to re-use.

Finally, effective problem solving strategies can also help avoid biases such as, Fixation(CB3), Convenience (CB6), and Subconscious Actions (CB7). For example, *convergent thinking exercises*–identifying a concrete solution to a problem–can help developers reach a (specific) solution quickly whereas, *divergent thinking exercises*–exploring multiple solutions to problems–can help developers identify an optimal solution from a set of alternatives. These prevent developers from fixating(CB3) on a single solution.

5.4.2 Tool Wishlist. Overall, participants felt that tool support to help overcome biases was lacking, and had difficulty naming any tools that they would use. Three participants (IP12, IP13, and IP15), however, recommended the following tools that they wished existed to help deal with each of the listed biases :

Fixation(CB3) can be reduced by IDEs that track developer actions and detect situations where a developer is "fixated". It can then prompt different actions. IP12 explained,

[12:44] *"... The IDE–if you change [code] and you always get the same error, it can say, hey, you have been do the same thing 5 times. But you always get the same error, maybe try something different?"*

Resort to Default(CB4) is a bias that developers will succumb to since it's a path of least resistance; as IP12 mentioned, "If there are default options, they'll just use it." and the way to overcome this problem would be via tool personalization and specification. He felt that tools that generate defaults that better match the current work context would be useful. For example, implementation of a high-level "intention" wizard that allows developers to "feed their intentions" into the wizard, which in turn then creates correct defaults and parameters relevant to the task.

Optimism(CB5): Tools that continuously run tests (and build scripts) in the background can counter this bias by identifying faulty changes that the developer might not have verified. IP12 recommended

[12:44] *"[the tool] could figure out, 'hey! this is [code area] where I could run the tests' and it'd run the tests for you without you having to doing anything.*

However, he warned that such tools can become intrusive and distracting to the developer if they continuously notify developers of failing tests.

Convenience(CB6) bias can be prevented by a tool that can identify sub-optimal code changes and recommend "clean" or "non-smelly" code. Not having "quick fix" changes can also help maintain backward compatibility and reduce technical debt. As IP1 explained,

[14:36] *"some tools that could identify a quick fix ... And then point out some of the problems that this particular fix will cause.*

Subconscious Actions(CB7)–based on misleading and recurrent environmental cues–can be prevented by annotating the severity of failures, exceptions, or results of flaky tests. IP12 mentioned annotating flaky tests,

[42:45] *"updating the cues to say, well, it's not a red, it's a blood red! Because there is a test that we know shouldn't fail is failing. A test that has never failed in the past 20 builds, did now!"*

Blissful Ignorance(CB8) can be avoided by tools that highlight a problem that appears similar to what the developer has experienced before and would otherwise ignore. Both IP15 and IP12 described a tool that allows developers to mark certain expected failures, such that the tool can notify them of other related failures.

IP15 recommended a different tool, which takes test results from the master branch and compares them with the current branch to identify areas that the developer should further investigate.

Memory(CB10) bias can be avoided by a tool that automatically identifies deprecated methods and recommends the relevant updated API functions, instead of the function that the developer remembered. IP13 mentioned:

[12:44] *"API code is evolving very frequently...and you don't know [the updated] methods...so one way to tell you like, 'hey, there's probably a better way of doing this.'"*

6 IMPLICATIONS

Our results indicate that cognitive biases frequently disrupt development, and compromise developers' problem solving abilities both in terms of task performance and time invested. Although developers currently deal with biases using a combination of standard and impromptu practices, there is a lack of tools that prevent or help recover from biases. Our findings have the following implications:

Implications for Researchers. *Onset and Effects of a Bias*: To proactively detect and mitigate effects of biases, research needs to identify when each of these biases manifests. This study reveals the tip-of-the-iceberg of the different effects biases can have on developers. Further studies are required to exhaustively understand specifically how individual biases affect development and how these different biases might interact.

Factors influencing biases: We noticed that biases were related to the type of task developers performed. For example, developers were more prone to fixation when debugging, which included small, frequent tweaks and rapid evaluation. Whereas, preconception biases were more prone to occur when developer implemented new features. Our interviewees also suggested that the extent to which a bias can affect a developer depends on experience (subconscious action - IP8), time constraints (convenience - IP9) and, in some cases, gender (optimism - IP13). Future studies are needed to identify how these factors might attenuate or exacerbate such biases.

Moreover, cognitive biases are not confined to software development tasks; In fact they are pervasive in disciplines involving complex human behavior and can impact different activities ranging from team decision-making to design acceptance. Other fields have researched methods to counter biases that involve either explicitly educating individuals to think in different ways [28], or create tools that intervene in decision making to reduce the manifestation of biases [1]. Exploring these methods across disciplines can uncover unconventional ways to deal with biases in software development.

Implication for Developers. Developers should be made aware that biases pose a significant threat to productive development, and perhaps are more pervasive than they realize. We synthesized a list of helpful practices (Table 6) that are expected to reduce the effect of cognitive biases. Some of these biases require an organizational level initiative. However, there are many practices that developers can individually employ on their own (e.g. divergence thinking, defensive programming etc.). Interviewees often discussed that such practices have long term benefits.

Implications for Tool Builders. Our interviews revealed that developers perceived a lack of tool support for dealing with biases. In Section 5.4, we identified various tool features that developers envisioned might help deal with with biases. Further, as developers currently rely on a combination of standard and improvised practices to deal with biases, these practices need better tool support for effective implementation. IP4, IP11, IP12 all recommended using very basic tools like personalized scripts, bullet journals, and even paper-pencil to create playbooks. Thus, our results represent an initial starting point for tool builders to actualize tools that help prevent and deal with frequently demonstrated cognitive biases.

7 THREATS TO VALIDITY

Like any field study, certain threats arise that might challenge our findings. We describe some of these threats and steps taken to mitigate them.

Although our observational findings are derived from a small number of participants from a single software development company, the startup nature of the company ensure variation among the participants in terms of tasks and tools. Our primary units of analysis were the 2084 participants' actions (as opposed to the individual participants). To bolster these observations we subsequently used a more diverse interview sample, which included participants from both large and small employers.

Observational studies are prone to confounds such as Hawthorne effect (or response bias), which can influence participant responses during our study [54]. We mitigate this threat by having only one researcher directly observe a participant during our study, but supported behind the scenes by a second observer. Additionally, like any other field study, we did not have complete control over the observed setting. However, we feel that this provides a higher degree of ecological validity, as we were able to observe participants reactions and performance 'in the wild', thereby producing a richer and ultimately more practical set of observation notes. While desirable, generalizability was not the main focus of this study [27], and we instead aim to present findings that provide transferable knowledge and insights that were not previously known.

Finally, given the large amount of data observed, there is always a concern regarding internal validity. To mitigate this threat we took explicit steps to maintain inter-rater reliability, and also implemented a well defined coding scheme and process [49].

8 CONCLUSION

In this paper, through a field study of 10 developers, we investigated both how often cognitive biases occur in the workplace, and how these biases impact development. Our results indicate that cognitive biases frequently disrupt development, and compromise developers' problem solving abilities like exploration, sense-making and contextual awareness. We conducted a complementary interview with 16 developers to support our observations, and in so doing compiled an initial set of practices and tools that developers currently use (or desire) to deal with cognitive biases.

Although the current work represents a strong initial effort towards better understanding cognitive bias, our understanding of biases and their effect in real-world development is still shallow. The current findings provide a useful starting point for future investigations, and future efforts at developing a deeper understanding of cognitive biases will help developers and researchers to implement more effective preventive practices, and guide tool builders in creating curated support.

ACKNOWLEDGMENTS

We thank the participants for their time and effort, Amelia Annel Leon and Yennifer Ramirez for their help with auxiliary analysis, and Rafael Leano for his support in conducting the study. This work is partially supported by the National Science Foundation under Grant Nos. 1560526 and 1815486.

REFERENCES

[1] David Arnott. 2006. Cognitive biases and decision support systems development: a design science approach. *Information Systems Journal* 16, 1 (2006), 55–78.
[2] Andrew Begel and Beth Simon. 2008. Novice software developers, all over again. In *Proceedings of the fourth international workshop on computing education research*. ACM, 3–14.
[3] Gul Calikli and Ayse Bener. 2010. Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 10.
[4] Gul Calikli, Ayse Bener, and Berna Arslan. 2010. An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. IEEE, 187–190.
[5] Gul Calikli, Ayse Basar Bener, and Farid Shirazi. 2013. Analyzing the Effects of Confirmation Bias on Software Development Team Performance: A Field Study during a Hackathon. In *39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013)*.
[6] Souti Chattopadhyay, Nicholas Nelson, Yenifer Ramirez Gonzalez, Annel Amelia Leon, Rahul Pandita, and Anita Sarma. 2019. Latent Patterns in Acttivities: A Field Study of How Developers Manage Context. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. ACM.
[7] Shirley SJO Cruz, Fabio QB da Silva, Cleviton VF Monteiro, Pedro Santos, and Isabella Rossilei. 2011. Personality in software engineering: Preliminary findings from a systematic literature review. In *15th annual conference on Evaluation & assessment in software engineering (EASE 2011)*. IET, 1–10.
[8] Brenda Dervin. 1983. *An Overview of Sense-making Research: Concepts, Methods, and Results to Date.* The Author.
[9] Premkumar Devanbu, Thomas Zimmermann, and Christian Bird. 2016. Belief & evidence in empirical software engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 108–119.
[10] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*. Springer, 285–311.
[11] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 402–413.
[12] Thomas Fritz and Gail C Murphy. 2010. Using information Fragments to Answer the Questions Developers Ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 175–184.
[13] D Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. 2006. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education* 9, 1 (2006), 1–8.
[14] Marko Gasparic and Andrea Janes. 2016. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software* 113 (2016), 101–113.
[15] Marko Gasparic, Gail C Murphy, and Francesco Ricci. 2017. A context model for IDE-based recommendation systems. *Journal of Systems and Software* 128 (2017), 200–219.
[16] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. 2012. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)* 19, 1 (2012), 5.
[17] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*. ACM, 91–100.
[18] Magne Jorgensen and Stein Grimstad. 2012. Software development estimation biases: The role of interdependence. *IEEE Transactions on Software Engineering*

38, 3 (2012), 677–693.
[19] Daniel Kahnema and Amos Tversky. 1972. Subjective probability: A judgment of representativeness. *Cognitive Psychology* 3, 3 (1972), 430 – 454. https://doi.org/10.1016/0010-0285(72)90016-3
[20] Mik Kersten and Gail C Murphy. 2006. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 1–11.
[21] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. 1265–1276.
[22] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2018. Data Scientists in Software Teams: State of the Art and Challenges. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1024–1038.
[23] Andrew J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 344–353.
[24] Joseph L. Fleiss. 1971. Measuring Nominal Scale Agreement Among Many Raters. *Psychological Bulletin* 76 (11 1971), 378–. https://doi.org/10.1037/h0031619
[25] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*. ACM, 492–501.
[26] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering* 39, 2 (2013), 197–215.
[27] Allen S Lee and Richard L Baskerville. 2003. Generalizing generalizability in information systems research. *Information systems research* 14, 3 (2003), 221–243.
[28] Jeanne Liedtka. 2015. Perspective: Linking design thinking with innovation outcomes through cognitive bias reduction. *Journal of product innovation management* 32, 6 (2015), 925–938.
[29] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 1449–1461.
[30] Richard E Mayer. 1992. *Thinking, problem solving, cognition . A series of books in psychology.* New York: WH Freeman/Times Books/Henry Holt & Co.
[31] André N Meyer, Laura E Barton, Gail C Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1178–1193.
[32] André N Meyer, Thomas Fritz, Gail C Murphy, and Thomas Zimmermann. 2014. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 19–29.
[33] Matthew B Miles, A Michael Huberman, Michael A Huberman, and Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. sage.
[34] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. 2018. Cognitive Biases in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering* (2018).
[35] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The Life-Cycle of Merge Conflicts: Processes, Barriers, and Strategies. *Empirical Software Engineering* (2019), 1–44.
[36] Nicholas Nelson, Anita Sarma, and André van der Hoek. [n. d.]. Towards an IDE to Support Programming as Problem-Solving.
[37] Rahul Pandita, Chris Parnin, Felienne Hermans, and Emerson Murphy-Hill. 2018. No half-measures: A study of manual and tool-assisted end-user programming tasks in Excel. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 95–103.
[38] Peter Pirolli. 1997. Computational Models of Information Scent-following in a Very Large Browsable Text Collection. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 3–10. https://doi.org/10.1145/258549.258558
[39] Peter Pirolli and Stuart Card. 2005. The Sensemaking Process and Leverage Points for Analyst Technology as Identified through Cognitive Task Analysis. In *Proceedings of International Conference on Intelligence Analysis*, Vol. 5. McLean, VA, USA, 2–4.
[40] Per Runeson and Martin Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14, 2 (2009), 131.
[41] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage.
[42] Donald Sharpe. 2015. Your chi-square test is statistically significant: Now What? *Practical Assessment, Research and Evaluation* 20 (01 2015), 1–10.
[43] David J. Sheskin. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures* (4 ed.). Chapman & Hall/CRC.
[44] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 23–34.
[45] Barry G Silverman. 1990. Critiquing Human Judgment using Knowledge-Acquisition Systems. *AI Magazine* 11, 3 (1990), 60.
[46] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging Among an Overabundance of Similar Variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 3509–3521.
[47] Keith E Stanovich. 2009. *What intelligence tests miss: The psychology of rational thought*. Yale University Press.
[48] Antony Tang. 2011. Software designers, are you biased?. In *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge*. ACM, 1–8.
[49] Sarah J Tracy. 2010. Qualitative quality: Eight "big-tent" criteria for excellent qualitative research. *Qualitative inquiry* 16, 10 (2010), 837–851.
[50] Amos Tversky and Daniel Kahneman. 1973. Availability: A heuristic for judging frequency and probability. *Cognitive Psychology* 5, 2 (1973), 207 – 232. https://doi.org/10.1016/0010-0285(73)90033-9
[51] Amos Tversky and Daniel Kahneman. 1974. Judgment under uncertainty: Heuristics and biases. *science* 185, 4157 (1974), 1124–1131.
[52] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.
[53] Gerald M Weinberg. 1971. *The psychology of computer programming*. Vol. 932633420. Van Nostrand Reinhold New York.
[54] Shaochun Xu and Václav Rajlich. 2005. Dialog-based protocol: an empirical research method for cognitive activities in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 10–pp.