Noname manuscript No.
(will be inserted by the editor)

Toward a Theory on Programmer's Block Inspired by Writer's Block

Belinda Schantong · Norbert Siegmund · Janet Siegmund

Received: date / Accepted: date

Abstract *Context:* Programmer's block, akin to writer's block, is a phenomenon where capable programmers struggle to create code. Despite anecdotal evidence, no scientific studies have explored the relationship between programmer's block and writer's block.

Objective: The primary objective of this study is to study the presence of blocks during programming and their potential causes.

Method: We conducted semi-structured interviews with experienced programmers to capture their processes, the problems they face, and potential causes. Subsequently, we analyzed the responses through the lens of writing.

Results: We found that among the programmer's problems during programming, several display strong similarities to writer's block. Moreover, when investigating possible causes of such blocks, we found a strong relationship between programming and writing activities as well as typical writing strategies employed by programmers.

Conclusions: Strong similarities between programming and writing challenges, processes, and strategies confirm the existence of programmer's block with similar causes to writer's block. Thus, strategies from writing used to resolve blocks should be applicable in programming, helping developers to overcome

Belinda Schantong
Chemnitz University of Technology
Germany
E-mail: belinda.schantong@informatik.tu-chemnitz.de

Norbert Siegmund
Leipzig University
Germany
E-mail: norbert.siegmund@cs.uni-leipzig.de

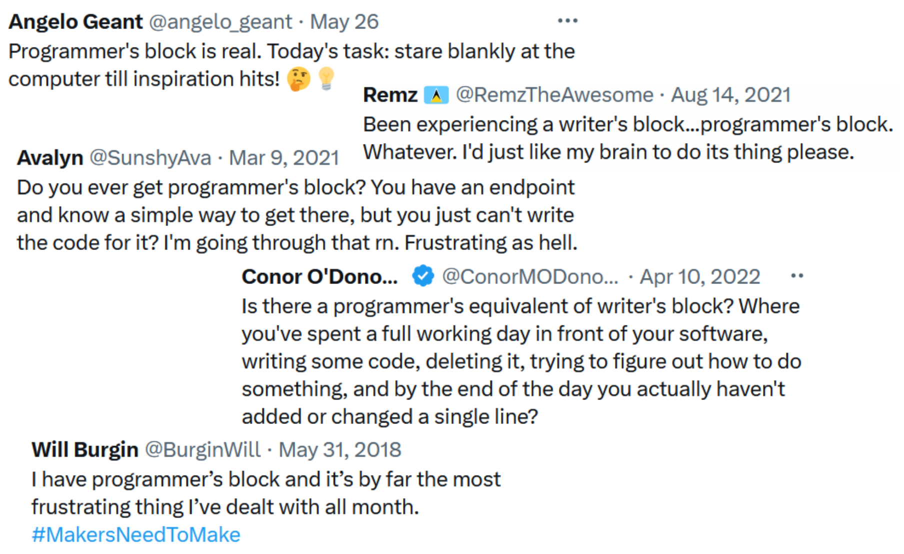
Janet Siegmund
Chemnitz University of Technology
Germany
E-mail: siegj@hrz.tu-chemnitz.de

phases of being stuck. Research at the intersection of both areas could lead to productivity gains through reduced developer downtimes.

Keywords Writer's block, programmer's block, qualitative study, professional programmers

1 Introduction

Software development is an inherently creative process (Groeneveld et al., 2021; Amin et al., 2018; Mohanani et al., 2017; Crawford et al., 2012) and similar to other creative processes, there is anecdotal evidence that programmers experience 'blocks', as illustrated in Figure 1: Programmers report lack of inspiration, frustration over their 'brain not doing its thing', and lack of meaningful progress. They appear to be capable programmers, too, indicated by them stating that they know ways to achieve their goal. They 'just can't write the code'. That implies that they are accustomed to being able to write code, and so that their condition cannot be attributed to a lack of basic programming skills.


Angelo Geant @angelo_geant · May 26
Programmer's block is real. Today's task: stare blankly at the computer till inspiration hits! 🤔💡

Remz @RemzTheAwesome · Aug 14, 2021
Been experiencing a writer's block...programmer's block. Whatever. I'd just like my brain to do its thing please.

Avalyn @SunshyAva · Mar 9, 2021
Do you ever get programmer's block? You have an endpoint and know a simple way to get there, but you just can't write the code for it? I'm going through that rn. Frustrating as hell.

Conor O'Dono... @ConorMODono... · Apr 10, 2022
Is there a programmer's equivalent of writer's block? Where you've spent a full working day in front of your software, writing some code, deleting it, trying to figure out how to do something, and by the end of the day you actually haven't added or changed a single line?

Will Burgin @BurginWill · May 31, 2018
I have programmer's block and it's by far the most frustrating thing I've dealt with all month.
#MakersNeedToMake

Fig. 1 Anecdotal evidence of blocked programmers.

The anecdotal evidence shows that programmers can get stuck on a task, which is often accompanied by negative emotions. Such reported times of reduced programming activity are supported by a large body of work on software developers' perceived productivity and their emotional states during work (Müller and Fritz, 2015; Graziotin et al., 2018, 2017b; Girardi et al., 2022; Sadowski and Zimmermann, 2019; Forsgren et al., 2021).

These blocks in programming seem to be common and taken seriously in practice, as professional developers share not only their experience, but also actively look to other creative disciplines for help, specifically to writing. Grey literature on the topic frequently draws the comparison to writer's block (Lee, 2015; Joury, 2020; Software, 2021; Kovacevic, 2021; Pasqualis, 2017), or even derives tips for programmers from literature on writing (Knight, 2017).

The instinctively perceived similarity of writing and programming is no coincidence: Already in the 1980s, Dijkstra suggested that good programmers also need good language skills (Dijkstra, 1982). Recent studies support this idea. For example, programming learning benefits from higher language aptitude (Prat et al., 2020) and technical reading training (Endres et al., 2021). Even in elementary school, language skills can help to acquire programming skills (Hassenfeld et al., 2020). Such results have been backed up by neuroimaging studies suggesting a link between programming and language skills, such that a continuous participation of language-processing areas has been demonstrated when programmers worked with source code (Siegmund et al., 2014a, 2017; Peitek et al., 2021; Floyd et al., 2017; Castelhano et al., 2019; Huang et al., 2019; Liu et al., 2020; Hongo et al., 2022; Lee et al., 2016; Medeiros et al., 2019).

Given the increasing amount of empirical evidence substantiating a link between language skills and programming skills, we explore whether the perceived programming blocks exist and can be related to writer's block. Since there is a rich body of scientific work on the diagnosis, analysis, treatment, and prevention of writer's block (Rose, 1984; Hjortshoj, 2001, 2019; Flaherty, 2005; Adams-Tukiendorf, 2008; Bastug et al., 2017; Rose and McClafferty, 2001; Wymann, 2021), and since blocks have been a subject of discussion in other creative disciplines, such as music (Scott, 2016) or visual art (Saraste, 2021; Gallay, 2013), we study whether similar conditions exist during programming, so that software-engineering practice may draw from this research on writing. Specifically, our goal is to find empirical evidence whether blocks exist and if so, what may be potential causes of these blocks.

1.1 Study Design

To study programmer's block, we derived a working definition of what we understand a block to be from existing definitions of writer's block in writing research, which we explain in depth in our background (section 2.1). We define programmer's block as follows:

Programmer's Block. It expresses the inability of writing or changing code despite being objectively able to do so:

- The programming task is known or given.
- Programming skills are sufficiently available (i.e., the programmer is profound in the current language).
- Programming infrastructure (i.e., IDEs, hardware, tools) is available and running.
- The programmer is willing and motivated to work on the programming task.

We then conducted interviews with 15 programming experts from several industrial contexts and analyzed the interviews based on this definition. One goal of our research was to critically evaluate whether our working definition is appropriate to capture this phenomenon.

1.2 Contributions

In a nutshell, we were successfully able to validate our working definition of programmer's block. Nearly all of the interviewed programming experts have reported that they have experienced blocks during their working processes. More importantly, they also reported that causes of such blocks often have their origin not during the activity of writing code, but in other phases of the development process, such as requirements specification or testing. This observation matches with causes of writers' blocks, because they can similarly originate from, for example, pre-writing activities, such as insufficient literature search. Thus, an investigation of programmer's block cannot be limited to the actual physical activity of writing code, but should entail the entire software development process.

We also found that the programming experts apply typical strategies from writing that help avoiding or resolving blocks. These results strengthen the connection of natural language processing and programming, giving more confidence to us and other researchers to use established strategies to overcome or avoid blocks. By drawing from guidelines and insights of writing research, we can train (beginning) programmers to apply useful strategies to avoid blocks and have an efficient programming process.

Next, we give an introduction to the state of the art of writing research and the definition of writer's block. We contribute a mapping of multiple writing processes, which we later use to align with software development activities to identify causes of programmer's block. Afterward, we provide an overview of related work in the context of software engineering, before explaining the conduct of our study (Section 3) and the results (Section 4).

2 Writing Research and Related Programming Research

Any writing is embedded in a *rhetorical situation* (Flower and Hayes, 1981; Hayes, 1996), and writers need to figure out how to communicate what they want to communicate within its constraints. The rhetorical situation describes the entire context of the writing process, including the social, physical, and individual situation, which all influence the writing process. For example, the process of writing differs between writing an application letter for a dream job and writing a birthday card to a friend. The *social situation* describes any persons related to the process, most notably the audience of the text. For example, writing a text for children is different than for a board of experts. In addition, the social situation includes the wider establishments in which the rhetorical situation is embedded, such as the institutional context within a university. Notably, a lot of writing research is done within universities and thus centers around academic and scientific writing. We consider this a benefit for our comparison, since scientific writing uses clear and explicit language, where each paragraph follows logically from the one before (Lindsay, 2020; Gustavii, 2017), and the goal of scientific writing is to inform the reader, so it should avoid using “flowery, ornate or obscure prose” (Lindsay, 2020). Similarly, programming naturally requires that information and the goals are explicitly expressed such that a computer can process it as well as a human reader can understand it, which is relevant, for example, when choosing identifier names (Hofmeister et al., 2017; Schankin et al., 2018; Beniamini et al., 2017). Thus, there are some parallels between programming and scientific writing, making scientific writing a good base to relate it to programming, even if one might think of prose first when the keyword ’writing’ comes up.

The *physical situation* then consists of the workplace, the already written text, and the medium used for writing. For example, handwriting is different from writing on a computer. Last, the *individual situation* is relevant, as the writer’s goals, motivation, knowledge, and physical capabilities affect the writing process. For example, writing the book they have always dreamed of is different than writing a mandatory report they find boring. All these parts of the rhetorical situation influence how writers engage with the writing process (Flower and Hayes, 1981; Hayes, 1996), and problems in any of these contexts can decrease a writer’s productivity. Some of the problems can be so severe that they lead to a writer’s block, which we discuss next.

2.1 Writer’s Block

Writer’s block describes a situation in which writers who are motivated and capable of producing a text, for some reason cannot access their skill (Rose, 1984). In other words, the actions of these writers do not lead to satisfying progress in the writing task (Hjortshoj, 2001).

This definition does raise the question of what ’capable’ means. Rose is specifically excluding writing problems from his definition of writer’s block which occur due to a ”lack of fundamental writing skills” (Rose, 1984). This refers to so called ’basic writers’, a term which historically denotes students whose writing skills were not considered ’college-ready’ (Shaughnessy, 1977). Hjortshoj keeps this part of the definition and explains that ”if I tried, without years of preparation, to write a publishable article in a field I know little about, such as biochemistry or astrophysics, I would struggle and fail, but the underlying problem would be my lack of ability, not a writing block.” (Hjortshoj, 2019) ’Capable’, thus, would mean that the writer has, or is at least perceived to have, the skills necessary to complete the writing task at hand. They might, however, misapply their knowledge and react with an inadequate strategy to a writing problem (Hjortshoj, 2001), or they might lack the adequate strategy to deal with a specific problem (Rose, 1984). One could say that they lack ’meta-skills’, or some knowledge about the writing process, while problems that hinder not-yet-capable writers concern grammar, syntax, or the contents they write about.

Thus, the root cause of blocks are misconceptions about the writing process (Rose, 1984): Writers try to adhere too rigidly to rules or methods they have learned about writing, for example, that 'all essays need to divide a thesis into three parts', or that 'each sentence needs to be perfectly crafted before moving on to the next' (Hjortshoj, 2019). Such misconceptions can relate to any part of the rhetorical situation, which means that there is not *the* writer's block, but that every case is unique (Rose, 1984). Nevertheless, some common patterns exist (Hjortshoj, 2019):

Conceptual problems Some blocks arise from faulty, incomplete, misunderstood, or non-existent concepts. Often, writers that suffer from blocks rely on strategies and plans that have worked for past projects, but which are not universally applicable and might be unsuitable for a new project. This often occurs when writers are transitioning to a new environment, for example, from high school to university (Wymann, 2021). Changing programming languages, frameworks, or architectural styles may be aligned to this kind of problem.

Some writers with blocks have conceptually unbalanced plans, where the concept does not support the actual content, for example, when writers think they are developing a theoretical framework, but actually conduct a case study (Hjortshoj, 2019). We can draw similarities when going from the design to the implementation phase during software development.

A further problem leading to conceptual blocks are misconceptions about a text's purpose: Writers with blocks might have misunderstood the purpose of the text which they are writing. For example, students often struggle with literature reviews, because they think their goal is to demonstrate their breadth of knowledge on the topic, when it is instead to explain and illuminate the significance of the following study (Hjortshoj, 2019). In software development, we see a lot of reports from failed, skipped, or insufficient requirements engineering, such that assumptions about the software are often incorrect or incomplete.

Performative problems When writers cannot use their full potential for completing a writing task, they can run into a performative block (Hjortshoj, 2019; Wymann, 2021). It may be caused by the misconception that the text, as it is being written, needs to meet expectations immediately that should only be set for the final product. This leads to a condition that is comparable to stage fright, where writers 'freeze up' in fear of their (imagined) audience. Although programmers are inclined to submit and refine code iteratively with the advent of continuous integration and agile development, we nevertheless operate under code and style guidelines, code reviews, and pressure towards high quality software, which could put programmers in a similar mindset.

Both types of blocks, conceptual and performative, are caused by a diverse set of misconceptions, and thus the solution to resolve blocks lies in uncovering and correcting these misconceptions (Rose, 1984; Hjortshoj, 2001,

Table 1 Comparison of writing activities separated by pre-writing, writing, and post-writing.

	Grieshammer et al. (2019)	Tompkins (1994)	Williams (2003)	Hermans and Aldewereld (2017)	Hassenfeld and Bers (2020
Pre-Writing	Orientation	Pre-Writing	Pre-Writing	Info gathering	Pre-Writing
	Collect materials				
	Structuring		Planning	Info selecting	
				Info structuring	
Writing	Drafting	Drafting	Drafting	Translating	Drafting
			Pausing		
			Reading		
Post-Writing	Revising	Revising	Revising	Stylizing	Revising
	Editing	Editing	Editing	Formatting	Editing
				Reflecting	Evaluating
		Publishing	Publishing		

2019). This requires thorough analysis of the blocked writer's *writing process* and needs to take into account the considerable inter-individual variance in *writing strategies* (Dengscherz, 2021; Sennewald, 2021; Wymann, 2021). This is why we need a thorough and sound qualitative design to identify whether such writing processes and writing strategies also exist for programming. This motivates how we designed our interview study (e.g., by not directly asking the participants about blocks).

2.2 The Writing Process and Writing Strategies

The writing process revolves around generating written output through reflection on input in the form of written media, sensual input, or one's own ideas (Hayes, 1996). It can be broken down roughly into *pre-writing* activities, the *writing* itself, and *post-writing* activities (Grieshammer et al., 2019; Tompkins, 1994; Williams, 2003; Hermans and Aldewereld, 2017; Hassenfeld and Bers, 2020), as shown in Table 1. The concrete activities mentioned vary slightly from author to author, depending on the focus of the description of the writing process, which often entails a didactic use, but with a focus on different writing situations.

Pre-writing activities are done before the actual writing. These can be divided in several ways. Grieshammer and others (Grieshammer et al., 2019) distinguish orientation (getting to know the rhetorical situation and choosing a topic accordingly) and collecting materials, since they focus on academic writing, where an adequate topic and the following research are of particular importance. Most authors (Grieshammer et al., 2019; Williams, 2003; Hermans and Aldewereld, 2017) recognize an activity where the gathered ideas and material are structured to plan their integration in the text. Hermans and Aldewereld (Hermans and Aldewereld, 2017) further distinguish between selecting the relevant information first before structuring it.

Writing activities revolve around producing text, that is, translating thoughts into written words. They are often referred to as *drafting* to differentiate them from the writing process as a whole (Grieshammer et al., 2019; Tompkins, 1994; Williams, 2003; Hassenfeld and Bers, 2020). It can be further subdivided and may contain drafting, or pausing and reading the text. So, not every pause is due to a block, but pauses are used to reflect on the current state of the text (Williams, 2003).

Post-writing activities entail working on the draft until it is finalized. They are usually subdivided into revising and editing. Revising means working on the clarity and style of the content. Editing means working on more mechanical criteria, such as grammar and spelling. These activities require writers to reflect on their work, taking the audience's perspective, possibly getting feedback from readers (Hermans and Aldewereld, 2017; Hassenfeld and Bers, 2020). Some authors also include publishing as a post-writing activity, as in sharing the text with its audience (Tompkins, 1994; Williams, 2003).

Although these activities roughly have an order, writing typically is not a linear process (similar to software development). Any of the activities can prompt almost any other activity. For example, while revising a chapter, writers might find that they still need to do more research, which might lead to re-structuring parts of the text. Some researchers also consider planning and revising to be their own applications of the writing process (Hayes, 2012).

How writers engage with these activities is referred to as their *writing strategy*. Most writers have a preferred strategy which they default to, and they might refer to themselves as *plotters* (starting with a structure) or *pantsers* (starting with drafting) (Hermans and Aldewereld, 2017). Between these two extremes, more strategies exist, such as *multiple-version writing* (creating multiple drafts instead of revising one), *writing by editing* (characterized by a short drafting process which quickly transitions into revising) and *syncretistic writing* (characterized by a particularly pronounced non-linearity of the process) (Sennewald, 2021; Wymann, 2021). Although we may easily find similar strategies for professional programmers in our own environment, we conduct an evaluation about the prevalence of such strategies in our interviews.

2.3 Related Work

Dedicated research to evaluate the existence of writer's block in the context of programming is sparse. To the best of our knowledge, only Hermans and Aldewereld evaluate the programming process through the lens of writing (Hermans and Aldewereld, 2017). By comparing writing and programming, they conclude that there are indeed similarities between both processes, for example, that gathering information in writing corresponds to defining the program objective, or that reflecting on the written text corresponds to compiling, testing, and debugging code. However, the authors did not conduct an empirical study, and do not mention writer's block. Moreover, the writing activities foundational to their mapping are based on a single resource, and the programming activities are based on a learning resource for basic programming that does not consider the wider software development process.

In a more philosophical line, Ciancarini and others describe software development as a story telling process (Ciancarini et al., 2020). In a systematic literature review (Ciancarini et al., 2023), they found that this idea has several applications, for example, as a tool for software engineers to describe and understand what a software system should do or to describe bugs in software (Devaney and Johnson, 2017), especially when the software development process is understood as a collaborative writing project (Bussell and Taylor, 2006). Other work focuses on programming as writing process. For example, Hassenfeld and Bers compared writing and programming as creative processes in the context of elementary school (Hassenfeld and Bers, 2020). Similar studies suggest teaching programming in school as language learning (Bers, 2019; Bers et al., 2023).

Already in 1989, Gantenbein suggested teaching programming as a process, since this approach had seen a lot of success in the teaching of writing (Gantenbein, 1989). Instead of viewing writing as simply putting words on paper, focusing on the process meant understanding writing as a "logical sequence of activities" (Gantenbein, 1989) that starts before any words are put on paper at all. Thus, Gantenbein suggested using a simplified version of the software life cycle as basic model for the programming process, to enable students to more easily understand problems during program development and to introduce strategies that help with different parts of the process. Similar ideas have been developed to reveal and teach the programming process more explicitly (de Raadt et al., 2009; Bennedsen and Caspersen, 2005). For example, LaToza and others developed a notation system to record explicit strategies in programming, to be used by learners and professionals alike (LaToza et al., 2020). Finally, there have been some studies observing the programming strategies used by novices, which have found differences in their approaches to planning that were independent of their performance (Whalley and Kasto, 2014). While we also examine programming as a process and what strategies programmers are using, we focus on the comparison to the writing process and how this helps us to identify and understand blocks.

Related to our work is program comprehension. Here, the link to writing research is within reading and writing a text to, for example, create an internal representation of a text (Hayes, 1996), much like an internal representation of the program (Schulte et al., 2010). While such internal processes of programmers and writers can be a promising further direction, we focus in this study on a very precise construct: the writer's or programmer's block. Moreover, aligning writing strategies and processes with software development and programming allows us to explore causes of blocks. Whether internal representations and the process of their creation of both fields actually align is another research question.

Another group of related approaches focuses on developer productivity, what it entails, and how it can be measured (Sadowski and Zimmermann, 2019). Sarkar and Parnin observe programmer fatigue, leading to decrease in productivity. Several indicators could hint at fatigue, such as error rates or warnings regarding software quality, which could be used to hint respective developers to take a break or switch to a different task (Sarkar and Parnin, 2017). Also, a bad night's sleep has been shown to decrease in performance (Fucci et al., 2018). The (un-)happiness of developers has also moved into focus of research, for example, when they do not have agency on how they complete their task (Meyer et al., 2021), or when they are stuck on a task (Graziotin et al., 2017b,a,c, 2018; Graziotin and Fagerholm, 2019). There is also work to determine whether developers are stuck. Müller and Fritz found that biometric sensors could detect whether developers were in flow or stuck (Müller and Fritz, 2015). Based on that, approaches to keep developers in the flow have been developed and found their way into practice (Züger et al., 2017). While this line of research also evaluates how productivity of developers is decreased, we are looking at one concrete aspect of reduced productivity, that is, the block, its causes, and its similarity to writer's block.

3 Study Design

We have outlined in Section 2 that problems in any part of the rhetorical situation can hinder the writing process. However, only a subset of these problems can be attributed to writer's block. This observation has consequences for designing a sound methodology for studying blocks in programming. First, we cannot simply ask whether the participants have experienced a block to avoid biasing the answers towards a certain outcome. Moreover, there is no clear definition of a block (yet), and asking for blocks without a clear definition could make participants think of scenarios that not necessarily resemble blocks (e.g., motivational problems). Second, a sound methodology should be able to explain or provide hints for causes of blocks, such that theories and actionables can be inferred and tested in future studies. Thus, we follow a qualitative approach by designing an interview following the processes of writing research, aligned to development activities. We use the integrated views of writing activities, summarized in Table 1 to map writing activities to development activities.

Our aim is to gain reliable, valid, and actionable insights from aligning writing research with development activities. To this end, we defined open-ended questions in a semi-structured interview to assess possible causes of problems during the development process (including blocks). This way, we aim to establish a relationship between writing processes and development processes to pin down the origin of a block's cause.

In summary, we state the following research questions:

RQ$_1$: *Are there blocks during code writing and editing that resemble writer's blocks?*

We aim to collect a large variety of problems during programming activities that possibly relate to writer's block. If our subsequent qualitative analysis can align these problems, we can find clear evidence for programmer's block.

RQ$_2$: *Do development activities map to activities of writing?*
To better understand the similarities and differences of writing and software development, we aim to identify a mapping of writing activities and software development activities. This enables us to pin down the root cause of previously identified blocks, if they exist.

RQ$_3$: *Do programmers use writing strategies?*
Writing research provides guidance for avoiding and resolving a writer's block. However, these guidelines are built on the current writing process (RQ2) and the writer's strategy (e.g., pantsing or plotting, cf. Section 2). To make a stance toward future reliable and tailor-made guidelines for programmers, we aim to determine whether these strategies also exist for programmers.

Next, we present the details of the study design. All material, including questions, participant data, and analysis procedure, is available at the project's web page (`https://anonymous.4open.science/r/WritersBlock-8C92`).

Philosophical Stance Our study design is driven by a constructivist epistemiological stance (Hoda, 2022), meaning that the knowledge that will be gained from our research is influenced by our existing knowledge, for example, on writing research and software engineering. This guides our design from the research questions we ask to the way we construct the materials, conduct the analysis, and interpret the results.

Material To answer our research questions, we constructed a semi-structured interview. We defined 33 questions, categorized into three parts. The first part contained 15 quick questions about the personal background of developers and their programming experience, which were taken directly from Siegmund et al. (2014b).

The second part consisted of up to 10 questions concerning the typical processes of the developers, to understand their programming strategies. The first question asked them to walk us through the process of a recent project of theirs, with potential follow-up questions to clarify the process, such as "What kind of project was it?", "How did you proceed at [a certain] part?", or "Is the process you described comparable to your other projects?". Not all of these questions needed to be asked if the initial description provided by the participant already contained these informations.

The third part contained up to 8 questions about problems that developers face in their processes. To explore the problems that might hint at or lead to a block, we first posed the question, whether the participants had ever discontinued a project. Such an event to us represents the most severe possible consequence of a block and is an outcome that we assumed the participants would have wanted to avoid, especially in a professional setting. Moreover, due to the severity of the event, we expect that participants can better recall it and according causes leading to the discontinuation. This sets the stage for less severe, but more frequent consequences of problems, from delays within projects to situations where the participants would simply feel 'stuck'. Such occurrences are likely more prevalent and provide insights into the diverse causes of blocks, leading to a more general picture of problems and blocks. Further questions were about the participants' strategies to overcome such problems. The final question was whether they think having ever experienced a programmer's block. We waited until the end of the interview to use this term in order to avoid any bias in the discussions beforehand, such that we are able to distinguish blocks from other problems.

Conduct We recruited participants using a mixture of strategies. We personally invited programming experts from our circles of acquaintances, some of whom offered to share the invitation with their own colleagues. Additionally, we sent out a call for participants via social media, specifically Twitter/X, and our university's mailing lists. If potential participants agreed to the interview, they were sent the participant information, which specified (i) the conduct and contents of the interview, (ii) our privacy policy, which contained all necessary legal information of how their personal data would be processed and protected , as well as (iii) the information about audio recordings. All participants gave written consent for all three parts (according to guidelines of our institution; no ethics approval is required for this type of study at our institution).

The interview took place either in a virtual, password-protected meeting room, or in an office at the university. The interviewer again asked for verbal consent to the audio being recorded during the interview before and after the start of the recording. The interview began with the questionnaire on programming experience, before moving to the semi-structured parts. Once all questions were answered, the participants were informed about the research goals, including that we are evaluating the existence of programmer's block and comparing it to writing and writer's block. The participants could then add additional thoughts if they wished to do so. The interview lasted between 18 and 56 minutes (mean: 33 minutes).

Table 2 Descriptive data of our participants. PY: Professional years; PE: Programming experience self evaluation (1: no experience; 10: master); PL: Programming languages.

ID	PY	PE (1-10)	# PL
P1	11	6	3
P2	3	5	3
P3	5	9	4
P4	1	9	11
P5	12	9	11
P6	12	9	5
P7	3	7	6
P8	7	5	2
P9	25	9	10
P10	4	6	4
P11	3	7	4
P12	8	8	3
P13	8	8	5
P14	19	8	6
P15	10	7	4
Ø	8,73	7,46	5,4

Participants We selected participants with an above average level of programming experience, so we could avoid problems that, according to the

definition of writer's block, would not
be considered a block when they rather originate from a lack of basic programming skill. A considerable level of programming experience was also important so that the participants would have sufficiently experienced problems during the development process that they could recount. To this end, we recruited 15 professional developers with considerable practical experience, resulting in a sample with an average of 8.7 years (median: 8 years) of professional programming experience. Table 2 gives an overview of our participants. With a mean self-estimated programming experience of 7.4 (on a scale of 1 to 10) and an average of over 5 programming languages that participants consider themselves proficient in, we have gathered an experienced group of participants that likely have experienced problems during the development process.

Analysis The interviews were transcribed, and the transcripts were then analyzed using open card sorting (Hudson, 2013). For each interview, we went over the responses of participants in a chronological order. We coded each statement belonging to a certain software development activity, for example, the statements "I say, okay, we can do this and this, we can't do that" (P_8) and "we had a meeting where we thought about, okay, what do we want to implement"(P_{10}) refer to the activity of defining objectives, and "I test all the time on the side" (P_2) and "[I] look if it works the way it should" (P_8) refer to testing and debugging.

We started with the three rough labels 'planning', 'implementing' and 'testing', which were refined during the analysis process into the more detailed labels 'defining objectives', 'research and preparation', 'design', 'writing code', 'testing and debugging', 'refactoring', and 'review'. The first and last author coded the first four interviews jointly, and once common ground on how statements can be categorized was established, the first author continued on her own, with the last author checking the assignment of codes to statements on a random subset. Statements that were unclear were discussed until reaching interpersonal consensus. In addition to the development activities, we marked statements that described a problem during the working process, such as lack of ideas, diversions, conflicts in the team, or technical problems.

To ensure that our sample size was sufficient, we calculated the saturation ratio according to Guest and others, determining a 'new information threshold', at which point no new information is provided by the interviewees (Guest et al., 2020). We started with a base of 4 interviews (as recommended), which covered 17 themes in total. We then used batches of 3 interviews in a sliding window approach of 1 interview to calculate the saturation ratio, that is, the ratio between new and already known themes. For our case, we had reached the new information threshold of 0 % after 12 out of the 15 interviews we conducted. Thus, with 15 interviews, we have a good base to capture the most relevant aspects of the programming experts' working processes and related problems.

This analysis formed the basis to answer the research questions. Each question required different steps of analysis going forward, which we explain next.

For RQ_1, we conducted a second iteration of tagging, focusing on the problem statements, grouping similar statements into categories. That is, we identified

(i) problems related to the working environment, for example, technical problems: "it has happened that my pc crashed the next day and for some reason windows completely decrypted this one source code file." (P_3)
(ii) problems related to programmers' motivation, for example, lack of interest in a project: "Some projects I canceled, mainly in the domain of graphics, because I found them to be too time-consuming and too far away from actual programming." (P_4)
(iii) problems related to the physical and cognitive resources, for example, tiredness ("after 16 hours in front of the code, there's a greater- or a less-than sign the wrong way around and you sit there and don't know why this isn't working"(P_8)), or lack of domain knowledge ("when I didn't have the domain-specific knowledge, for example, in the beginning of working with an EEG."(P_4))
(iv) problems related to the processes during software development, such as unclear requirements: "It was a pricing system and we were going to improve pricing, whatever that meant. (laughs) And we could never define what that meant, right?"(P_9)

In a final iteration, we mapped the statements to the development activities, so that we can understand what causes of problems occur in which parts of the development process.

For RQ_2, we mapped the activities stated by the programmers to the activities of the writing process (cf. Table 1). To this end, we used the descriptions of the development activity provided in the interviews and decided within the author team whether a mapping can be established, is out of scope of the study context, or has no counterpart in a writing activity. For example, the development activity of defining objectives was described by statements, such as P_{10}'s: "We had a meeting where we thought about, okay, what do we want to implement." This bears similarity to the pre-writing activity 'orientation' (in which the writer chooses the topic to write about; cf. chapter 2.2), so we will map it to the pre-writing activity of orientation. Based on such comparisons, the first author prepared a first mapping, including markings for unclear cases. The third author evaluated this mapping. In case of disagreement (less than 10 %) or unclear cases, the entire team discussed each case in detail until reaching interpersonal consensus.

In a similar vein, for RQ_3, we mapped the processes described by the programmers onto the writing strategies introduced in Section 2.2, namely *spontaneous* (the 'pantser'), *planning* (the 'plotter'), *multiple versions*, *editorial* (writing by editing), and *chaotic* (syncretistic writing). Similarly to RQ_2, the first author provided an initial mapping that was evaluated and refined by the entire team in discussions.

4 Results and Discussion

Next, we review our results and discuss possible implications.

4.1 RQ1: Are there blocks during code writing and editing that resemble writer's blocks?

To answer RQ_1, we examine the problems that programmers encounter in their working processes and compare them to the different kinds of writer's block introduced in Section 2.1. To identify problems, we asked the programmers whether they have ever discontinued a project and what lead to that decision, or whether they have encountered significant delays in their working processes and what lead to those delays.

In total, we evaluated 76 stated problems, which are caused by *external factors*, such as technology, people, the physical working environment, and existing code, as well as *internal factors*, such as motivation and knowledge. We followed a conservative approach to identify blocks. Problems that can be attributed to a lack of motivation or a lack of skill are not considered blocks according to the definition of writer's block (Rose, 1984) (cf. Section 2.1). Additionally, we do not consider a problem as a block that occurred due to a temporary state of tiredness, for example, a lack of concentration at the end of a long day. Finally, we do not consider problems with a clear tangible cause in the working environment as blocks, for example, technical problems, such as a computer failing to work, or a delay due to a different component not being finished yet. Note that all of these factors could still interact with a block, for example, a block could lead to a lack of motivation, which in turn could aggravate a block. Following this logic, we excluded problem statements related to the working environment (29 statements, 38 %), the programmers' motivation (4 statements, 5 %), and the programmers' physical or cognitive resources (12 statements 16 %) from further analysis. The remaining 31 statements (41 %) could be related to blocks. Upon further investigation of these 31 statements, we were able to identify potential causes for 29 of them.

Figure 2 provides an overview of reported problems (blue) as well as identified blocks (red). We align both to the development activities in which the causes of such problems have been reported in the interviews. In essence, we found concrete blocks as well as other sources of delay throughout different development activities. Interestingly, we observed that (i) most causes of blocks have been reported at the intersection of activities, and (ii) there are only few causes of reported blocks during actual code writing. In the following, we will qualitatively evaluate the results and provide a first interpretation for these observations.

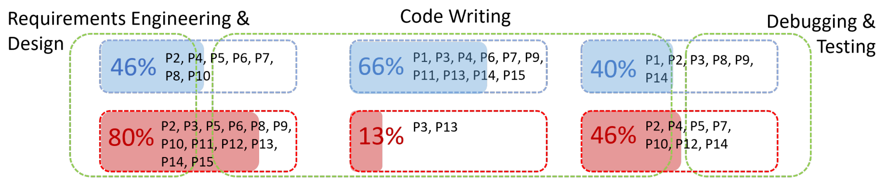

Requirements Engineering & Design
Code Writing
Debugging & Testing
46%
P2, P4, P5, P6, P7, P8, P10
66%
P1, P3, P4, P6, P7, P9, P11, P13, P14, P15
40%
P1, P2, P3, P8, P9, P14
80%
P2, P3, P5, P6, P8, P9, P10, P11, P12, P13, P14, P15
13%
P3, P13
46%
P2, P4, P5, P7, P10, P12, P14

Fig. 2 Reported causes of problems related to blocks and delays with respect to the development activities where they originate from. Red areas indicate the number of participants who reported problems related to blocks, including the participant IDs. Blue areas depict further problems related to delays that are not associated with blocks. The percentage depicts how many interviewees out of our 15 have encountered causes of blocks and other productivity problems in the respective development activities.

4.1.1 Causes at the intersection of design and code writing

80 % of the participants reported cause of programmer's blocks at the intersection of requirements engineering/design and code writing. The cause that has been described most often is unclear and/or repeatedly changing requirements. As P_{10} puts it: "What I always found most challenging is when clients themselves were not sure about what they wanted exactly, and it turned out during the development process that they actually wanted something different." Statements like these were made by P_3, P_5, P_8, P_9, P_{10}, P_{12}, and P_{13}. Programmers, like writers, need to develop a concept of the software they are working on (Petre, 2010; Robins, 2019), and unclear requirements or repeated changes can hinder the development of a concept, causing blocks in the subsequent development activities.

Similarly, we interpret statements in which the participants describe that requirements can be too big or too complex, or that a project can grow much bigger than initially anticipated (P_2, P_3, P_6, P_{13}, P_{14}). The problem and its consequences are explained well by P_3: "Sometimes, there is a really big requirement or a giant idea, but I just don't ... I know what I *want* to do, but how to represent that in code? I have been stuck on that sometimes and have even discarded projects, because I could not come up with an implementation in my head." Similar problems have been observed in writing, where writers are blocked because their topic is too broad or reveals itself to be too broad in implementation, even if it seemed to be focused when presented as a coherent outline (Hjortshoj, 2019).

Other problems at this intersection of activities include solutions or concepts that do not work, either due to some unknown reason or explicitly due to bad design (P_2, P_6, P_9, P_{10}, P_{11} P_{14}), as well as programmers not being able to create a solution at all (P_6, P_{15}).

4.1.2 Causes at the intersection of code writing and testing

Seven participants (46 %) reported problems that were related to the activities of testing and debugging (P_2, P_4, P_5, P_7, P_{10}, P_{12}, P_{13}). Interestingly, the programmers do not seem to think of bugs themselves as causes that actively hinder the process; only when their established strategies at handling them fail, developers start seeing bugs as problems. Exemplary in this regard is a case from P_5, where their software crashes “right in that phase of the booting process, where you get no information from the firmware anymore, but do not have a driver yet. There, I am just blocked, because I have no idea how to get in there, to find out what is going wrong.” In line with the idea of a conceptual block, this could mean that, in the case of such difficult bugs, the programmer’s concept of the program is flawed or incomplete: P_5’s concept is not extensive enough to completely understand how their software and hardware are interacting. P_5’s statement also illustrates one possible strategy to counteract such situations: Using the assistance of technology to expand the mental model. Tools, such as debuggers and other visualizations, can assist programmers in developing mental models (Petre, 2010), however in the case of P_5, this strategy does not suffice, causing the block.

Statements from other participants further cement the idea that a lacking concept could cause blocks during debugging. P_{14} states: “It can be difficult to tell, is this a bug or is it intended behavior that just does not fit my use case.” P_{10} describes a somewhat different, yet similar situation: “You think, it [a function] should be doing X, but it does XY.” In these cases, programmers expect the code to do something which it does not actually do, meaning their concept of the code does not match reality, making it a flawed concept. The programmers must then correct their concept, but if their strategies to do so fail, they might run into a block.

4.1.3 Unknown causes

Only two participants, P_3 and P_{13}, describe blocks that are constrained to the activity of writing code. P_3 describes a situation where what they are doing is just not working: “It often happens that I try to implement things, it doesn’t work, it doesn’t work, it doesn’t work, it doesn’t work, and then at some point, I will have introduced an error.” The fact that they specify towards the end that this state will inadvertently lead to errors suggests that what is ‘not working’ is not yet referring to errors in the code. Thus, the problem lies in the process itself. P_{13} describes that there are instances where they simply do not know how to translate their thoughts into code: “I sit in front of [the task] for a day, knowing what the end result should be and having a rough idea of what should happen, but I experience a real block and don’t know how to formulate it in source code.” Both statements mirror descriptions of behavior of writer’s with blocks, with P_{13} directly mentioning blocks. However, they do not provide any further hints towards possible causes of the blocks, which still might be hidden in a different activity of the development process. Thus, we consider these statements to be symptoms of blocks, but cannot know what their causes are without further research.

4.1.4 Performative causes

Notably, we did not find concrete evidence of performative causes for blocks. However, performative causes might exist as the statement of P_6 indicates: "It is important to see that [...] having no fear of change and of revising a decision and to review and draw conclusions from that, that you are not afraid of that." P_6 explains the importance of not being afraid of making changes to the code, thus implying that they might have encountered such fear in the past. A fear or unwillingness to change code would force a programmer to write perfect code on the first try, which could lead to a similar destructive perfectionism that occurs during writing in the form of performative blocks. However, the lack of statements describing such problems suggests that they are less prevalent in software development compared to writing, at least for our participants. Already this insight demands further research: Is fear of making changes a thing or not? If so, does it spawn at large companies with heavy code reviews, in open-source systems with social pressure, or only for development methodologies that hinder refactoring, such as when deployed in avionics systems or embedded in physical devices?

4.1.5 Summary of discussion and answer to RQ1

In section 1.1, we defined programmer's block as the inability of writing or changing code, even though there is a given task, the programmer has sufficient skills in the programming language, the infrastructure needed for programming is available, and the programmer is motivated to work on the task. This was derived from Rose's definition of writer's block introduced in section 2.1, who defined writer's block as the inability to start or continue writing, but not due to the lack of basic writing skills or motivation.

Through analyzing the problem statements from our participants with this definition in mind, we were able to distinguish between problems that were not related to blocks, and problems that were potentially related to blocks and could not be ascribed to any other category of problems. Thus, nearly all participants (14 out of 15) reported incidents that clearly fall under our definition of programmer's block, meaning our analysis shows strong empirical evidence for the existence of programmer's block based on our definition.

Moreover, when further analyzing the statements that were potentially related to blocks, we found hints that blocks likely originate in similar places, namely at the intersection of the development activities that make up the software development process, and that they have similar causes. We especially found hints towards conceptual causes.

Answer $\mathbf{RQ_1}$: Yes, programmers encounter blocks that resemble writer's block. Most cases originate at the intersection of different development activities. We found that most causes can be related to conceptual blocks, but found little indication of performative blocks.

4.2 RQ2: Do development activities map to activities of writing?

To answer RQ_2, we analyzed the activities during the working processes that developers described, and compare each of them with the definition of the writing activities (cf. Section 2.2). In essence, the participants talked about different activities during development, including requirements analysis, design, code writing, testing, and refactoring, which we mapped through analysis to the writing activities (cf. Section 2.2). To this end, we use the mapping of writing activities across the literature from Table 1 as a basis, and then complement it with the development activities recounted by the participants. We present the mapping in Table 3. Next, we discuss the development activities as the developers described them in the interviews, the mapping to the writing activities, and the process of how we arrived at this mapping. We use the partition of the activities into pre-writing activities, writing activities, and post-writing activities, as used in writing research (cf. Section 2.2), to structure the discussion. However, this does not mean that either writing or developing software are linear processes.

4.2.1 Pre-writing Activities

Orientation → Defining Objectives Most participants begin their programming projects by defining their objectives. They describe how they talk to clients and define the task and requirements (P_1, P_3, P_5, P_8, P_{11}, P_{13}). P_8 summarizes their process as follows: “I say [to the customer], okay, we can do this and this, we can’t do that.” This also entails estimating time and effort a task will take and planning the workflow (P_1, P_3, P_6, P_9). In private projects, the processes are less specified, such that the activity is more about having an idea and defining objectives on how to implement it (P_4 P_9, P_{10}). We summarize these activities under the term ‘defining objectives’, to be inclusive of these processes that go beyond just requirement engineering

We map the activity of defining objectives to the writing activity of ‘orientation’, during which writers figure out what they want or need to write about (cf. Section 2.2). To this end, they generate ideas (Williams, 2003; Tompkins, 1994) or choose a topic within the requirements of their context (Grieshammer et al., 2019; Tompkins, 1994). To do this, they need to analyze and understand said context and requirements (Hermans and Aldewereld, 2017; Tompkins, 1994).

Thus, both processes have the same function of setting the basic conditions for the project (even though they might look different on the surface). The rhetorical situation of writing tasks can vary greatly, including writing in a team or receiving the writing task from a third party. Similarly, a single programmer working on a passion project alone undergoes a complete development process just as much as professional developers in a team, which was also demonstrated by our participants (e.g., P_3, P_4, P_{14}). For similar reasons, Hermans and Aldewereld (2017) map the writing activity of ‘gathering

Table 3 Mapping between the writing activities introduced in Table 1 and the development activities. The IDs in the 'Mapping' column indicate which participants described the corresponding programming activity.

	Grieshammer et al. (2019)	Tompkins (1994)	Williams (2003)	Hermans and Aldewereld (2017)	Hassenfeld and Bers (2020)	Mapping	Activity
Pre-Writing	Orientation	Pre-Writing	Pre-Writing	Information gathering	Pre-Writing	$P_1, P_2, P_3, P_4, P_5, P_6, P_8, P_9, P_{10}, P_{11}, P_{12}, P_{13}$	Defining objectives
	Collect materials					$P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{13}, P_{14}$	Research, code reading
	Structuring		Planning	Information selecting		$P_1, P_2, P_3, P_4, P_7, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}, P_{15}$	Design
				Information structuring			
Writing	Drafting	Drafting	Drafting	Translating	Drafting	$P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}, P_{15}$	Writing code
			Pausing				
			Reading				
Post-Writing	Revising	Revising	Revising	Stylizing	Revising	$P_3, P_5, P_6, P_{11}, P_{14}$	Refactoring
						$P_1, P_2, P_3, P_5, P_6, P_7, P_8, P_{10}, P_{11}, P_{12}$	Testing (& debugging)
	Editing	Editing	Editing	Formatting	Editing		
		Publishing	Publishing			$P_1, P_2, P_3, P_5, P_6, P_8, P_9, P_{10}, P_{12}, P_{13}, P_{14}$	Review

information'[1] with the development activity of 'defining program objectives': "When starting a new project, assignment, or exercise in writing or programming, the first step is setting of the goals." (Hermans and Aldewereld, 2017). Thus, our data is in line with their mapping.

Gather Information/Collect materials → Research and Code reading A notable activity in the working processes described by our participants was research. For example, they research how to accomplish certain tasks, such as P_4: "I just started programming by looking at how to create an app frontend, how to do things with it. So, I first looked and clicked around a bit, explored until I had a design principle that looked good." They also need to figure out the tech-stack they want to use (P_6, P_7), or even compare the approaches of business competitors (P_{13}). This includes a lot of code-reading, as P_7 explains: "Since you cannot dictate a complete architecture yourself, you usually spend a significant amount of time initially examining how existing code is structured. [...] This currently takes up a considerable part of the 'research time'." Understanding the code they want to expand is only one scenario where researching existing code is necessary. The participants also look at prior work they might want to reuse (P_8, P_9), or search for existing sample programs online (P_{14}).

The mapping of researching activities of our participants to the writing activity of 'gathering information' Hermans and Aldewereld (2017) or 'collecting materials' (Grieshammer et al., 2019) is straight forward. This writing activity is necessary for writers to inform their developing text (cf. Section 2.2). It can overlap with the activity of orientation, if the information the writers seek is about the context of a writing project (Hermans and Aldewereld, 2017), but it often entails more than that. Writers need to do research on their topic (Hassenfeld and Bers, 2020) and gather their own data (Grieshammer et al., 2019).

These activities are comparable, because both, writers and programmers, usually do not work with an empty canvas, but use available information, which will considerably inform their further process. As P_7 explains: "The further preparatory process mostly depends on the framework, because each of them has its peculiarities and dictates its own structure to some extent." Just as the framework dictates the structure of the software, the collected research and data informs the structure of a scientific thesis (Grieshammer et al., 2019) and even fiction writers derive structure from outside information. For example, when writing a detective story, writers might research how detective work is done in the real world to give the story a realistic framework.

Interestingly, the researching activities described by our participants have mostly been absent from the existing models of the programming or software development processes, such as the ones used by Hermans and Aldewereld

[1] It should be noted that (Hermans and Aldewereld, 2017) use the term 'gathering information' both for the activity of setting goals and the activity of collecting materials and discuss them together.

(2017) and Hassenfeld and Bers (2020). The suggested processes by Bennedsen and Caspersen (2005) merely allude to the idea by mentioning the existence of online documentation, while Gantenbein (1989) at least recognizes that a programmer needs to ask themselves the question of 'what tools are available' and that this could include looking into predefined library routines. Additionally, the importance of the process of reading programs as a part of the process of writing programs is often addressed (see for example Schulte et al. (2010)), but not necessarily how it pertains to this researching process and its wider influences on the entire development process. Overall, however, the wider activity of researching during software development remains quite understated in the literature, and is mostly just implied.

Structuring and Planning → Design Participants think a lot about the structure of their software, though they call it 'design' (P_4) or 'architecture' (P_{11}). The programmers need to think of how exactly features will look and how they interact with each other (P_{10}). P_{13} explains how discussing a feature results in precise plans, up to already thinking in pseudocode: "It is often the case that, during these appointments, it is already very, very deeply discussed, and the feature is, I would say, already implemented a little bit in imaginary pseudocode." Similarly, P_5 describes that they devise a plan for the implementation already when they formulate a ticket: "I usually make the plan while I'm working on the ticket[...]. And thus, I only get assigned the tickets when I already know how I'm going to do it." Some programmers take notes or create checklists (P_7, P_{12}, P_{15}), whereas for others, these processes are mostly in their head, and they only reluctantly externalize their thoughts if necessary: "Sometimes, if it is really complicated [...], then one sketches it down sometimes for oneself a bit stepmotherly." (P_2). This activity is closely interlinked with the previously mentioned activities of defining objectives and research and preparation. P_4 directly derives their design from their research, and P_5 explains that, to create a plan, they need to do a lot of code reading first. P_1 also mentions how designs can be developed during the meetings in which requirements are discussed.

In writing, once information and ideas have been gathered, they are organized to develop the structure of the text. (Grieshammer et al., 2019; Hermans and Aldewereld, 2017). The plans writers make range from rough outlines down to the words which they want to write (Hassenfeld and Bers, 2020).

Where the programmers structure features and the software's architecture, writers structure arguments or storylines (Hermans and Aldewereld, 2017). The level of plans has a similar range, as well. Where some writers create outlines, some programmers create class diagrams (e.g., P_4), and where some writers formulate sentences in their head (Hassenfeld and Bers, 2020; Sennewald, 2021), programmers formulate imaginary pseudocode (P_4, P_{13}).

There is also a similar divergence where sometimes programmers like to keep it all in their head and externalize their plans only if really necessary (P_2, P_7), whereas other times, they write themselves clear plans and to-do lists (P_7, P_12, P_{15}). The same can be observed with writers (Sennewald, 2021).

Thus, not only are the activities of structuring and design similar, writers and programmers also do it in similar ways.

4.2.2 Writing Activities

Drafting/Translating → Writing Code Our participants need to implement their designs into a programming language. Interestingly, writing code is mentioned only briefly in the interviews. The participants all imply it: They 'build' something (P_9, P_{12}, P_{14}), 'implement' what they have described before (P_1, P_6, P_10), simply 'program' (P_2), 'write (code)' (P_5, P_7, P_{11}), or 'add functionality' (P_3, P_8), but they do not elaborate how. The process seems to come naturally to them, as P_5 describes: "At the end of the day, when it comes to implementation, it's really just [...] I have to write a bit of code here, a bit of code there, but that's all it is."

In writing research, the activity of translating abstract concepts into natural language and externalizing them as words on the page is defined under the terms 'drafting' Williams (2003) or 'translating' (Hermans and Aldewereld, 2017) to distinguish it from the overall writing process. For our purposes, we chose the term 'writing code' for this development activity, since the term does not lead to the same ambiguity in the context of software development.

That experienced programmers hardly consider worth mentioning the part of the process that outsiders would identify as the actual 'programming' is somewhat surprising. However, it might well be that experienced programmers hardly notice the transcription process as a significant part of the work anymore. This was also believed in early work on writing research, such that experienced writers have automated this transcription process to a degree where it could be safely ignored (Hayes, 2012). However, later studies in writing research revealed that transcription indeed still does compete with other activities for cognitive resources, which is why it is part of state-of-the-art models (Hayes, 2012). While cognitive load theory has been applied to software development as well, especially in the context of programming education (Robins et al., 2019), it would be interesting to dig deeper into this phenomenon in relation to programming experts. For example, it would be interesting to investigate whether the process of translation into a programming language can be mostly ignored for experienced programmers, as our interviews would somewhat imply, or whether the translation process requires more cognitive resources from less experienced programmers by comparison.

4.2.3 Post-writing Activities

While for the first four activities, the mappings of development activities to writing activities was rather straightforward, for the post-writing activities, the picture is fuzzier. What we are able to unquestionably identify from our interviews is that there are several 'post-code-writing activities', that is activities that continue working with code that has been written beforehand. As

P_{11} puts it in their description of their workflow: "It's basically always programming, writing a test, and then refactoring." However, a clear mapping of these development activities to the post-writing activities is difficult to establish. In our discussion of these activities, we address the reasons for this and the parallels we still encountered.

Revising & Editing → Testing (and Debugging) & Refactoring Our participants mention testing often in conjunction with writing the code(P_3, P_7, P_{11}). P_2 explains: "I program that [...] and just test all the time on the side [...] and see if what is happening is what should be happening". Similar statements were made by P_3, P_7 and P_{11}.

Some make an explicit difference between formal and informal testing, such as P_{10}: "I write the code, informally test it for myself, so in that case maybe I click through it, and then I write proper test cases, so unit tests or something like that, which then cover all cases." Some of the developers who work in companies also explain that they are supposed to use a test-first approach, but in practice often prefer writing code first (P_7, P_{10}, P_{11}).

Notably, none of the participants explicitly mentioned debugging the code when they described their usual process, they all only spoke of testing. We only assume that they would also debug their code, should their tests find any errors.

It could be argued that, through this close connection between writing code and testing it, and through the clear distinction that is made between formal and informal testing in some cases, that at least the informal part of testing does not need to be interpreted as its own development activity. Instead, it could be interpreted as being a part of the 'code writing' activity. However, since writing code and testing are usually explicitly called out as two separate activities (e.g., P_2 and P_{10}), we also treat them as such.

Even if not considered one and the same activity, it could still be argued that the most optimal mapping between testing and any writing activity would be between testing and 'drafting/translating', arguing that there is no testing activity in text writing. However, there is a clear temporal separation between writing the code and testing it, since, in order to test code, code must have been written[2].

Looking at the post-writing activities then, it could be argued that testing, or maybe more clearly debugging, could be mapped to the activity of 'editing' (editing in writing means working on mechanical concerns of the text, such as grammar and spelling). In writing instruction, this is usually recommended at the very end of the process, since the draft of a text can still be understood even with some errors; thus, it is not productive to edit early (Grieshammer et al., 2019). However, the fact that it is a known problem that writers tend to edit too early (Rose, 1984) means that in practice, it is often done at other points in the writing process.

[2] It is possible to write tests before writing the code to be tested, but not to do the actual *activity* of testing the code.

The argument for mapping testing and debugging to the editing activity is that both activities entail correcting mechanical errors, which is the reason why Hassenfeld and Bers (2020) map 'debugging mechanical errors' and 'editing'. However, our participants do not actually talk about debugging, let alone about debugging mechanical errors. They test "if what is happening is what should be happening"(P_2), or "if it works the way I imagined it"(P_8). Thus, the activity of testing described by our participants does not map to the activity of editing, and while developers might still be fixing mechanical errors during either code writing or testing (so there is no data against the mapping from Hassenfeld and Bers), this is not the activity which our participants are describing.

The most appropriate mapping might then be to the activity of 'revising'. Revising the text means working on high-level concerns, so that the text matches the plan (Grieshammer et al., 2019; Williams, 2003) and meets the needs of the audience (Tompkins, 1994).

The goal of 'matching the text with the plan', or in this case the code with the plan, has clearly been expressed by the descriptions of our participants. This does, however, leave us with the question where refactoring comes into play here.

P_{11}, describes refactoring as the third part of their core workflow of 'programming, testing, refactoring'. They do not explain further what they mean by refactoring, but refactoring is also mentioned by other participants, though it is less present compared to the other activities. P_6 explains that they understand refactoring as "changing architectural decisions", thus re-designing the program. P_3 defines refactoring as "you look at a piece of code a second time and improve it". P_5 describes a project where they had to do a lot of refactoring, because the code "looked all topsy-turvy". P_{14} does not call it refactoring, but states that they rewrite the code once the functionality is accomplished: "Once it's working, I try to redo it properly." P_3 also talks a lot about 'bad' code and about tools that help to alarm the developer if code is 'unreadable'. Thus, there does seem to be a development activity aimed at making the code 'better' or 'more readable', and it is sometimes explicitly called refactoring, which is why we also use the term to describe this activity.

'More readable' entails the question of more readable for whom? This would have to be a human audience, because a computer does not care about readability. This dichotomy has been discussed by Videla (2018) and has led them to the conclusion that code essentially has two distinct audiences: The machine readers and the human readers. Since one of the goals of the revising activity in writing is to have the text meet audience needs (which would include readability), this would lead us to the conclusion that the activity of 'refactoring' is also best mapped to the activity of revising.

This means that there are two distinct revising activities in software development, depending on which audience the revision is targeting: The activity of testing and debugging matches the code with the plan and thus targets the machine audience, while the activity of refactoring targets the human audience of the code.

Evaluating/Reflecting → Review When explaining how they 'test' what they have programmed, some of the participants did not refer to testing as mentioned in the paragraph above, but to code reviews, where they give the code to other people for feedback. P_8, for example, explains: "When the basic framework is completed, we move on to the initial tests, where I let my colleagues try it out and ask, 'Here, do you like this, does it work like this?' " P_{10} separates testing and review more clearly: "So, I have implemented my feature, tested it, and was satisfied. Then it went to one of the other two. They conducted a review, focusing on code quality, whether the code is understandable, as well as overall testing with the entire system." Thus, even though P_8 calls what they do an 'initial test', it seems to be closer to what P_{10} describes as review. The fact that P_8 jumps directly from implementing the code to the step of review underlines again how closely connected writing the code and testing are, but it also reveals the overlap between testing and reviews.

The writing activity that maps best to this activity of reviewing would be 'evaluating' (Hassenfeld and Bers, 2020) or 'reflecting' (Hermans and Aldewereld, 2017). Writers reflect on and evaluate their text to see whether what they have produced matches what they had planned and whether it meets all objectives (Williams, 2003; Hermans and Aldewereld, 2017; Hassenfeld and Bers, 2020).

In a review, the program is inspected from the audience's point of view, but the participants reveal that there can be several audiences to a program. On the one hand, the audience are code reviewers in the classical sense of 'code quality', for example, whether code is 'understandable' (P_1, P_2, P_9, P_{10}, and P_{14}). On the other hand, the audience can be customers or users, as P_3 explains: "In this review, the guy who originally set the requirements is present. He examines the implementation of the user story, and if something is not right, we have to discuss it thoroughly." Similarly, P_1, P_2, P_9, and P_{14} report that they get feedback from users or take the users' point of view when they review code within the team.

The existence of these two audiences is also reflected in the participants' answers to the question of whom they are thinking of when working on their project, which was specifically included to reveal whether coding has an audience, and which some participants also answered with the user and some with other programmers who have to maintain the code later on. Apart from revealing that there are several different audiences to programs, we also see that there are several layers to review in the software development process. This can also be found in writing: While some researchers put the activity of reflection towards the very end of the writing process within the post-writing activities, others include reflective activities within the writing activities (notably Williams (2003), cf. 2.2). And of course, writers also have the possibility to get feedback from collaborators or test audiences.

Thus, there is some nuance to this comparison, especially when it comes to the post-writing activities: In writing, the activities of reflection, revision, and editing all strive for the same goal of making the text understandable for

Table 4 Overview of found writing strategies and the scenarios in which they have been reported.

Strategy	Participants	Contexts
Spontaneous	P_2, P_3, P_6, P_9, P_{11}, P_{13}, P_{14}	private projects, exploration of frameworks, exploration of ideas and solutions
Planning	P_2, P_3, P_4, P_5, P_7, P_8, P_{10}, P_{12}, P_{13}, P_{15}	traditional software development, user-story based, ticket systems
Multiple versions	P_2, P_3, P_4, P_9, P_{13}, P_{14}	prototyping, exploration of alternative approaches, reworking broken approaches
Editorial	P_3, P_5, P_6, P_7, P_8, P_9, P_{13}, P_{14}	iterative development, refactoring
Chaotic	P_4, P_{11}	parallel development of multiple projects, freedom to try new things

the audience. In software development, on the other hand, programmers have to keep in mind the needs of at least three distinct audiences: The machine who interprets the code, the programmer who maintains the code, and the user who runs the code.

Answer RQ$_2$: Yes, development activities can be mapped to writing activities.

4.3 RQ3: Do programmers use writing strategies?

To answer RQ$_3$, we analyzed the interviews for indicators of writing strategies (cf. Section 2). We found that the strategies that exist for writing are also used in software development, and that the participants switch strategies to adjust to different problems and environments. In Table 4, we present the writing strategies that participants described, including the scenarios in which they are used. In the following, we explain each strategy in detail.

4.3.1 Spontaneous

The spontaneous strategy is often mentioned in context of private programming projects, as P$_9$ describes: “If I’m building something just for myself, it tends to be very unstructured. Because usually I’m doing it for my enjoyment, right? [...] Usually that’s not in architectural diagrams.” P$_9$ derives joy from the process of coding itself and is less interested in careful planning. P$_{11}$ describes similar differences between their development process at work and in private: “[In private] I take it less seriously (laughs). [...] I don’t think through the architecture that much, but just sit down and start. [...] The deeper you are inside the project, the more of the architecture shows itself." The notion that the structure of the project reveals itself to the writer instead of being carefully crafted is typical for the spontaneous strategy (Sennewald, 2021).

Interestingly, this also occurs in professional projects: "Decisions about the architecture unfold themselves once you start writing" (P_6). P_{14} also describes their project similarly: "It gradually developed itself into a simple object-oriented design. Nothing too complex and I didn't plan it beforehand, but it just so happened that that approach worked best."

P_{13} uses the spontaneous strategy for exploration: "If you want to try something out, I think that jumping right into it is the only right approach, because it takes away a lot of blocks. So if you just want to look at a new framework, then go! On with it!" Thus, also in professional contexts, the spontaneous strategy has its place.

4.3.2 Planning

The planning strategy often occurs in the context of classical software engineering projects, in which programmers work in a team with a clearly defined workflow, be it sequential or agile. P_5 and P_{13} both talk about how they already have very clear plans in their mind, down to the level of 'imaginary pseudocode' (P_{13}), before they even get a task assigned. P_7 and P_{10} like to lay out a plan in comments before they start coding. P_8 explains their planning as follows: "I think about that beforehand, what sub-functions I need to have that main functionality, and then, well, it's four or five lines of code per sub-function." P_{15} also explains how they create their plans: "I look at the feature: What should happen next? What is the basic idea? With that, I build a road map, and then I just have to implement that." Thus, the planning strategy also exists, and it seems to best fit a well-defined workflow.

4.3.3 Multiple Versions

P_{14} explains that their projects often contain complete rewrites, sometimes up to four versions. P_{14} prefers creating a minimal viable product for the first idea, because "if it turns out to be garbage, it is easy to discard such a throwaway-prototype". Thus, the idea that early versions are disposable is an advantage, which P_3 describes as well: "I can just start and write dirtily, but productively. And once I see that the idea has some substance and could become a great project, I start over." P_{13} also applies this strategy in a professional setting: "It is very much possible that, once you test your very early mega-alpha-version, you realize that the way data is handled is completely wrong or does not fit the use case, and then it can happen that you tear it all down and start over." Here, the strategy might not be the most desired path, but is still applied to avoid more severe problems.

However, there are also cases in which this strategy is explicitly forbidden. P_2 mentions that "[w]e program in a way that has to work on the first try. If anything goes wrong, well, then we didn't have enough time." P_7 states on a similar case: “The motto at the time was: ‘There have already been millions invested in development time. It needs to be further developed; it must not be rewritten!’ ” In the end, the entire project was cancelled due to the poor quality of the codebase. It might be that in these cases, allowing multiple versions could actually be beneficial to decrease development time. Thus, the multiple versions strategy exists, but is not always preferred.

4.3.4 Editorial

This strategy is similar to iterative development, which we found in the interviews. P_9 works explicitly with a ‘minimum viable product’: “Once [...] we knew that it could work, [...] we iterated and added features.” P_{14} also creates a ‘smallest possible prototype’ for a feature, focusing on getting it to work before expanding on it step by step. P_8 too describes their process as creating the rough core structure of the program, before editing, reworking and adding functionality to that core. In contrast to the *multiple versions* strategy, the participants here do not ‘tear it all down’ or ‘start over’, but they gradually add to their rough first draft.

P_5 describes an editorial process from a case which was rather unusual for them: “I inherited that code, which was a mess. And thus, there basically arose a big refactoring, where a lot of new code was produced without much of a plan beforehand, but was really put together piece by piece.” Here, they use an editorial approach, developing the code ‘piece by piece’, ‘without much of a plan’, even though planning would usually be the strategy they prefer. Thus, programmers can adjust their strategy to the task at hand.

4.3.5 Chaotic

This strategy is quite pronounced with P_4. At an early point in the interview, they described their process as ‘back and forth programming’. When asked what that meant, they explained that they did not really undergo a design process, but thought of a feature they found interesting, went to research how to do such a feature and then implemented it, before thinking of the next thing. When they ran into trouble, they just skipped that part for the time being and worked on something else.

P_{11} also tends to do several things in parallel, but with a more focused strategy to avoid task switching too often: “It’s a phenomenon when working on several projects at once, that you, for example, focus on one thing, then you have to wait for tests, and then you need to flip that switch in your head, to get into the other project with a completely different problem at hand, where you need to immerse yourself in that again”, they explain. Thus, we find some evidence that the chaotic strategy exists, but it might also be seen as a necessity to increase productivity.

4.3.6 Summary

We found that programmers use strategies that correspond to five major writing strategies. Their usage depends on different factors, such as working environment (e.g., planning is favored in larger teams), project goals (e.g., the spontaneous strategy is used to explore new frameworks), whether a project is seen by other people (e.g., the multiple versions approach for private projects), and the size of a project (e.g., small projects can more easily be written with the spontaneous strategy). This matches with the results of Whalley and Kasto (2014), who observed that beginner programmers use different planning strategies based on their personal preferences, but that especially higher performing students were also able to adjust their strategy based on the complexity of the task. With our sample of more experienced developers, we found a higher variance of viable strategies and how they were applied. Therefore, our results show that experienced developers can adapt their strategy to changes in the development process, thereby avoiding loss of productivity caused by inefficient strategies.

Answer RQ$_3$: Yes, programmers use known writing strategies in their development process and adapt their strategy to the context of the development task.

5 Implications

Our interviews revealed that (i) programmer's block exists and has similar causes as writer's block, (ii) writing processes and software development share common activities, (iii) writing strategies are also applicable for software development, and (iv) programmers can select a suitable strategy depending on their needs. This allows us to infer recommendations to prevent and resolve programmer's block from writing research.

Education and Writing Strategies Programmers use established writing strategies, but the same is not necessarily true for less experienced programmers. Writing researchers found that preventive instruction ameliorates many writing problems (Hjortshoj, 2019), and the same might also count for programming learning. Preventive instruction would mean to teach realistic processes and strategies to programming learners that leave room for the variance of working situations that developers might encounter later. Contrasting this to current software-engineering education, we are not aware that similar strategies are taught or even the presence of programmers' block is discussed (not to mention how to resolve one). Writing strategies can also benefit programming instructors. As Whalley and Kasto (2014) point out, planning is usually seen as the preferred strategy. But it can be difficult to account for students who are reluctant to do that. Knowledge about writing strategies enables writing instructors to better understand those students and help them broaden their range of strategies (Sennewald, 2021), which could also be relevant for programming.

Thus, we recommend further studies to carve out explicit writing strategies to teach them early on in undergraduate curricula. This may well integrate to the software life cycle, as we found that blocks occur at intersections of phases. Hence, we see large potential for productivity gains with proper teaching.

We have also found some insights about the software development process which are mostly absent from current models suggested for teaching, especially that researching activities are important during software development. It might be beneficial for programming novices to learn more explicitly that researching is a relevant and normal part of the process.

(Automated) Early Detection of Programmer's Block The existence of blocks that we identified motivates new methods and tools for reliably detecting them early on to reduce unproductive time. Since each block is different, we need to analyze the current situation of the development process. Once a developer recognizes a productivity drop, guiding questions to the developer might be helpful to clarify the situation for deriving the kind of block. Here, we can draw again from writing research, for example (Hjortshoj, 2019): *When did the block first occur, in what stage of the project? What kinds of projects could you complete more easily, and what makes this one more difficult? Do you have a central goal and a viable plan for the structure?* With more dedicated research on programmer's block, these questions could be better tailored to the exact blocks that programmers encounter. A long term goal of this research could be to detect and address programmer's block early with tool support (e.g., by analyzing code contributions or keystrokes). With the advent of coding assistants, such as ChatGPT or CoPilot, AI-assistance may not solely focus on code generation, but may be used to state such questions to the programmer to quickly detect blocks and collect information about the cause.

Resolution guidelines for detected programmer's block Once a programmer's block has been detected, different options may resolve this non-productive phase. First, encouraging programmers to switch to a different strategy represents an adaption of a successfully proved resolution in writing (Hjortshoj, 2019; Sennewald, 2021). Programmers already do that, and explicitly reminding them to switch strategies may have a huge impact on productivity.

Second, providing ample opportunity to talk to other developers about one's projects and potential issues may help, as writing research found that it can be difficult to identify and revise one's own habitual strategies that are not working (Hjortshoj, 2019). Although such advice does not sound surprising at first, strongly encouraging it (rather than having an incidental successful coffee break) might prove to be especially effective in case of blocks.

Third, tackling the root cause of a block is also a suggested resolution strategy (Rose, 1984). The aforementioned guiding questions are key to analyze the cause of a block and to select a suitable strategy to resolve it. For instance, the conversation with an AI stating such questions could quickly point to an unclear or too large user story, preventing the programmer from implementing it. The suggested resolution may then be revising the story first instead of forcing the implementation of it.

6 Threats to Validity

Like for all empirical studies, there are threats to validity. First, **external** validity is naturally limited by our sample. To mitigate this threat, we computed an information threshold to identify whether our sample saturated in the identified problems during software development. However, since we are using a constructivist approach, our results are naturally limited to similar contexts in which they were conducted (Hoda, 2022). With regard to our sample, this means that the selection of participants is influenced by our experiences and our means of recruitment. While we did achieve some diversity in the backgrounds of our participants regarding their experience and working environments, and while we made sure to achieve a saturated sample, this study is not able to cover all possible perspectives. Nevertheless, since we focus on creating new knowledge to develop a theory of programmer's block, this does not diminish our results. Finally, our insights are currently constrained to programming experts. We can also imagine a follow up study, widening the external validity to programming beginners: It would be interesting to evaluate the take of beginning programmers on the actual translation process, and whether they simply lack experience to run into an actual block.

To ensure **internal** validity without biasing participants, we defined the questions for a semi-structured interview beforehand, starting from general to specific questions. We carefully designed the questions according to the state-of-the-art in writing research regarding writer's block and writing activities. The interviewers followed these questions for each interview to avoid influencing the participants by mentioning blocks. Since neither of the terms 'block', 'programmer's block', 'writer's block', or 'writing text' at all were mentioned during the interviews until the very last question (unless the participants brought them up themselves), it is unlikely that the interviews were influenced by the participants potentially knowing about discussions of programmer's block or comparisons between programming and writing.

Since our method of qualitative data analysis is inherently subjective, we cannot just get rid of it completely, so our conclusions are influenced by our knowledge of writing research and the software development process. With our constructivist epistemological stance (cf. Section 3), the results are a product of said influences. In this case, the first author who did the majority of the initial coding for the analysis already had a background in writing counselling and was viewing the data through that lens. We see this background, however, as a strength rather than as a possible threat to validity. We did, however, reduce subjectivity as far as possible, by following a structured approach. We regularly exchanged our understanding of the data within the author group, and compared and adjusted our mappings accordingly.

7 Conclusion

Programmer's block indeed exists beyond anecdotal reports. Experienced programmers can get stuck on their task and are unable to complete it, even though they have everything they need. This is comparable to the concept of writer's block. But even beyond that, we found comparable processes and strategies in writing and software development. From these similarities, we can draw actionables on how to detect and resolve blocks, as well as better educate developers to learn proper coping strategies in case of unproductivity. In the future, it is interesting to further compare writing and software development, for example, to identify whether a developer is currently experiencing a block and if so whether tailored strategies help in solving or avoiding blocks.

Declarations

Conflict of Interest

No funding was received to assist with the preparation of this manuscript and the authors do not have any financial interests to declare. Janet Siegmund is part of the journal's editorial board. Norbert Siegmund has published work with advisory board member Tim Menzies within the past three years.

Data Availability

All data is available at the project's Web site: `https://anonymous.4open.science/r/WritersBlock-8C92`. Once the manuscript is published, we provide a persistent link on Zenodo.

Consent and Ethics Approval

All participants were received a participant information detailing the contents, conduct and purpose of the interview, a privacy policy detailing how their personal data would be processed and protected, as well as information on how the audio of the interviews would be recorded. All participants gave their written consent by signing these three documents before participating in the interview, and again gave verbal consent immediately before the interview started. All of our proceedings are in line with the GDPR and guidelines from the German Research Foundation. No ethics approval is required for this type of study at our institution.

Acknowledgements Special thanks to all the participants in the interviews.

References

Adams-Tukiendorf M (2008) Overcoming Writer's Block in an MA Seminar. Zeitschrift Schreiben (8):1–10, URL https://zeitschrift-schreiben.ch/2008/#adams

Amin A, Basri S, Hassan MF, Rehman M (2018) A Snapshot of 26 Years of Research on Creativity in Software Engineering - A Systematic Literature Review. In: Kim KJ, Joukov N (eds) Mobile and Wireless Technologies 2017, Springer, Singapore, Lecture Notes in Electrical Engineering, pp 430–438, DOI 10.1007/978-981-10-5281-1_47

Bastug M, Ertem IS, Keskin HK (2017) A phenomenological research study on writer's block: causes, processes, and results. Education + Training 59(6):605–618, DOI 10.1108/ET-11-2016-0169, URL https://doi.org/10.1108/ET-11-2016-0169

Beniamini G, Gingichashvili S, Orbach AK, Feitelson DG (2017) Meaningful Identifier Names: The Case of Single-Letter Variables. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp 45–54, DOI 10.1109/ICPC.2017.18, URL https://ieeexplore.ieee.org/abstract/document/7961503

Bennedsen J, Caspersen ME (2005) Revealing the programming process. In: Proceedings of the 36th SIGCSE technical symposium on Computer science education, pp 186–190

Bers MU (2019) Coding as Another Language: A Pedagogical Approach for Teaching Computer Science in Early Childhood. Journal of Computers in Education 6(4):499–528, DOI 10.1007/s40692-019-00147-3, URL https://doi.org/10.1007/s40692-019-00147-3

Bers MU, Blake-West J, Kapoor MG, Levinson T, Relkin E, Unahalekhaka A, Yang Z (2023) Coding as another language: Research-based curriculum for early childhood computer science. Early Childhood Research Quarterly 64(3):394–404, DOI https://doi.org/10.1016/j.ecresq.2023.05.002, URL https://www.sciencedirect.com/science/article/pii/S0885200623000571

Bussell B, Taylor S (2006) Software development as a collaborative writing project. In: Abrahamsson P, Marchesi M, Succi G (eds) Extreme Programming and Agile Processes in Software Engineering, Springer-Verlag, Berlin, Heidelberg, pp 21–31

Castelhano J, Duarte I, Ferreira C, Duraes J, Madeira H, Castelo-Branco M (2019) The Role of the Insula in Intuitive Expert Bug Detection in Computer Code: An fMRI Study. Brain Imaging and Behavior 13(3):623–637, DOI https://doi.org/10.1007/s11682-018-9885-1

Ciancarini P, Masyagin S, Succi G (2020) Software design as story telling: Reflecting on the work of italo calvino. In: Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ACM Press, pp 195–208, URL https://doi.org/10.1145/3426428.3426925

Ciancarini P, Farina M, Okonicha O, Smirnova M, Succi G (2023) Software as storytelling: A systematic literature review. Computer Science Review 47:100517:1 – 100517:21, URL `https://doi.org/10.1016/j.cosrev.2022.100517`

Crawford B, de la Barra CL, Soto R, Monfroy E (2012) Agile software engineering as creative work. In: 2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE), IEEE, pp 20–26, DOI 10.1109/CHASE.2012.6223015

Dengscherz S (2021) Considering individual and situational variation in modeling writing processes. In: Gustafsson M, Eriksson A (eds) Negotiating the Intersections of Writing and Writing Instruction, The WAC Clearinghouse, pp 165–194, URL `https://doi.org/10.37514/INT-B.2022.1466.2.06`

Devaney K, Johnson J (2017) Storytelling as a Key Enabler for Systems Engineering. International Council on Systems Engineering 27(1):894–907, URL `https://doi.org/10.1002/j.2334-5837.2017.00401.x`

Dijkstra E (1982) How do we Tell Truths that Might Hurt? In: Selected Writings on Computing: A Personal Perspective, Springer-Verlag, pp 129–131

Endres M, Fansher M, Shah P, Weimer W (2021) To Read or to Rotate? Comparing the Effects of Technical Reading Training and Spatial Skills Training on Novice Programming Ability. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), ACM, p 754–766, DOI 10.1145/3468264.3468583, URL `https://doi.org/10.1145/3468264.3468583`

Flaherty AW (2005) The Midnight Disease: The Drive to Write, Writer's Block, and the Creative Brain. Houghton Mifflin Harcourt

Flower L, Hayes JR (1981) A cognitive process theory of writing. College Composition and Communication 32(4):365–387

Floyd B, Santander T, Weimer W (2017) Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 175–186, DOI 10.1109/ICSE.2017.24, URL `https://doi.org/10.1109/ICSE.2017.24`

Forsgren N, Storey MA, Maddila C, Zimmermann T, Houck B, Butler J (2021) The SPACE of Developer Productivity: There's more to it than you think. Queue 19(1):Pages 10:20–Pages 10:48, DOI 10.1145/3454122.3454124, URL `https://dl.acm.org/doi/10.1145/3454122.3454124`

Fucci D, Scanniello G, Romano S, Juristo N (2018) Need for Sleep: The Impact of a Night of Sleep Deprivation on Novice Developers' Performance. IEEE Transactions on Software Engineering 42(42):42

Gallay LH (2013) Understanding and treating creative block in professional artists. Alliant International University

Gantenbein RE (1989) Programming as process: a "novel" approach to teaching programming. ACM SIGCSE Bulletin 21(1):22–26

Girardi D, Lanubile F, Novielli N, Serebrenik A (2022) Emotions and Perceived Productivity of Software Developers at the Workplace. IEEE Transactions on Software Engineering 48(9):3326–3341, DOI 10.1109/TSE.2021.3087906, conference Name: IEEE Transactions on Software Engineering

Graziotin D, Fagerholm F (2019) Happiness and the productivity of software engineers. In: Sadowski C, Zimmermann T (eds) Rethinking Productivity in Software Engineering, Apress, Berkeley, CA, pp 109–124, DOI 10.1007/978-1-4842-4221-6_10, URL https://doi.org/10.1007/978-1-4842-4221-6_10

Graziotin D, Fagerholm F, Wang X, Abrahamsson P (2017a) Consequences of unhappiness while developing software. In: 2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion), IEEE, pp 42–47, DOI 10.1109/SEmotion.2017.5

Graziotin D, Fagerholm F, Wang X, Abrahamsson P (2017b) On the Unhappiness of Software Developers. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, ACM, pp 324–333, URL https://doi.org/10.1145/3084226.3084242

Graziotin D, Fagerholm F, Wang X, Abrahamsson P (2017c) Unhappy developers: Bad for themselves, bad for process, and bad for software product. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, pp 362–364, DOI 10.1109/ICSE-C.2017.104

Graziotin D, Fagerholm F, Wang X, Abrahamsson P (2018) What happens when software developers are (un)happy. Journal of Systems and Software 140:32–47, DOI 10.1016/j.jss.2018.02.041, URL https://www.sciencedirect.com/science/article/pii/S0164121218300323

Grieshammer E, Liebetanz F, Peters N, Lohmann B (2019) Zukunftsmodell Schreibberatung: Eine Anleitung zur Begleitung von Schreibenden im Studium. Schneider Verlag Hohengehren, Baltmannsweiler, Germany

Groeneveld W, Luyten L, Vennekens J, Aerts K (2021) Exploring the Role of Creativity in Software Engineering. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), IEEE, pp 1–9, DOI 10.1109/ICSE-SEIS52602.2021.00009

Guest G, Namey E, Chen M (2020) A simple method to assess and report thematic saturation in qualitative research. PloS one 15(5):e0232076

Gustavii B (2017) How to write and illustrate a scientific paper. Cambridge University Press, 3rd Edition

Hassenfeld ZR, Bers MU (2020) Debugging the Writing Process: Lessons From a Comparison of Students' Coding and Writing Practices. The Reading Teacher 73(6):735–746, URL https://doi.org/10.1002/trtr.1885

Hassenfeld ZR, Govind M, de Ruiter LE, Bers MU (2020) If You Can Program You Can Write: Learning Introductory Programming Across Literacay Levels. J of Information Technology Education: Research 19:65–85, DOI 10.28945/4509

Hayes JR (1996) A new framework for understanding cognition and affect in writing. In: Levy CM, Ransdell S (eds) The Science of Writing. Theories, Methods, Individual Differences, and Applications, Routledge, pp 1–27

Hayes JR (2012) Modeling and Remodeling Writing. Written Communication 29(3):369–388, DOI 10.1177/0741088312451260, URL https://doi.org/10.1177/0741088312451260

Hermans F, Aldewereld M (2017) Programming is Writing is Programming. In: Companion to the first International Conference on the Art, Science and Engineering of Programming, ACM, pp 1–8, URL `https://doi.org/10.1145/3079368.3079413`

Hjortshoj K (2001) Understanding Writing Blocks. Oxford University Press on Demand

Hjortshoj K (2019) From Student to Scholar. A Guide to Writing through the Dissertation Stage. Routledge

Hoda R (2022) Socio-technical grounded theory for software engineering. IEEE Transactions on Software Engineering 48(10):3808–3832, DOI 10.1109/TSE.2021.3106280

Hofmeister J, Siegmund J, Holt DV (2017) Shorter identifier names take longer to comprehend. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, Klagenfurt, Austria, pp 217–227, DOI 10.1109/SANER.2017.7884623, URL `http://ieeexplore.ieee.org/document/7884623/`

Hongo T, Yakou T, Yoshinaga K, Kano T, Miyazaki M, Hanakawa T (2022) Structural neuroplasticity in computer programming beginners. Cerebral Cortex 33:5375–5381, DOI 10.1093/cercor/bhac425

Huang Y, Liu X, Krueger R, Santander T, Hu X, Leach K, Weimer W (2019) Distilling Neural Representations of Data Structure Manipulation using fMRI and fNIRS. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 396–407, URL `https://doi.org/10.1109/ICSE.2019.00053`

Hudson W (2013) Card Sorting. In: Foundation ID (ed) The Encyclopedia of Human-Computer Interaction, Interaction Design Foundation

Joury A (2020) How to overcome coder's block. when you're scared of your console. URL `https://towardsdatascience.com/how-to-overcome-coders-block-51ece9dafe00`

Knight H (2017) What Writers Can Teach Programmers | HackerNoon. URL `https://hackernoon.com/how-to-solve-programmers-block-18363c040656`

Kovacevic A (2021) How to beat coder's block – five tips to help you stay productive. URL `https://www.freecodecamp.org/news/how-to-beat-coders-block-and-stay-productive/`

LaToza TD, Arab M, Loksa D, Ko AJ (2020) Explicit Programming Strategies. Empirical Software Engineering 25(4):2416–2449, DOI 10.1007/s10664-020-09810-1, URL `http://arxiv.org/abs/1911.00046`, arXiv:1911.00046 [cs]

Lee J (2015) 5 ways to beat programmer's block right now. URL `https://www.makeuseof.com/tag/5-ways-beat-programmers-block-right-now/`

Lee S, Matteson A, Hooshyar D, Kim S, Jung J, Nam G, Lim H (2016) Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis. In: Proc. Int. Conf. Bioinformatics and Bioengineering (BIBE), IEEE, pp 350–355

Lindsay D (2020) Scientific writing = thinking in words. CSIRO Publishing, 2nd Edition

Liu YF, Kim J, Wilson C, Bedny M (2020) Computer Code Comprehension Shares Neural Resources with Formal Logical Inference in the Fronto-Parietal Network. eLive 9:e59340, URL https://doi.org/10.1101/2020.05.24.096180

Medeiros J, Couceiro R, Castelhano J, Branco MC, Duarte G, Duarte C, Durães J, Madeira H, Carvalho P, Teixeira C (2019) Software Code Complexity Assessment Using EEG Features. In: International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), IEEE, pp 1413–1416, DOI 10.1109/EMBC.2019.8856283

Meyer AN, Barr ET, Bird C, Zimmermann T (2021) Today Was a Good Day: The Daily Life of Software Developers. IEEE Transactions on Software Engineering 47(5):863–880, DOI 10.1109/TSE.2019.2904957

Mohanani R, Ram P, Lasisi A, Ralph P, Turhan B (2017) Perceptions of Creativity in Software Engineering Research and Practice. In: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, pp 210–217, DOI 10.1109/SEAA.2017.21

Müller SC, Fritz T (2015) Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, pp 688–699, URL https://doi.org/10.1109/ICSE.2015.334

Pasqualis L (2017) Understanding and overcoming coder's block. URL https://dev.to/lpasqualis/understanding-and-overcoming-coders-block-5m6

Peitek N, Apel S, Parnin C, Brechmann A, Siegmund J (2021) Program Comprehension and Code Complexity Metrics: An fMRI Study. In: Proc. Int. Conf. Software Engineering (ICSE), IEEE, pp 524–536, URL https://doi.org/10.1109/ICSE43902.2021.00056

Petre M (2010) Mental imagery and software visualization in high-performance software development teams. Journal of Visual Languages & Computing 21(3):171–183, URL https://doi.org/10.1016/j.jvlc.2009.11.001

Prat CS, Madhyastha TM, Mottarella MJ, Kuo CH (2020) Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages. Scientific Reports 10(1):1–10, DOI https://doi.org/10.1038/s41598-020-60661-8

de Raadt M, Watson R, Toleman M (2009) Teaching and assessing programming strategies explicitly. New Zealand 95

Robins AV (2019) Novice Programmers and Introductory Programming. In: Fincher SA, Robins AV (eds) The Cambridge Handbook of Computing Education Research, Cambridge University Press, pp 327–376

Robins AV, Margulieux LE, Morrison BB (2019) Cognitive Sciences for Computing Education. Learning Sciences Faculty Publications 22, URL https://scholarworks.gsu.edu/ltd_facpub/22

Rose M (1984) Writer's Block: The Cognitive Dimension. Carbondale (Ill.)

Rose M, McClafferty KA (2001) A Call for the Teaching of Writing in Graduate Education. Educational Researcher 30(2):27–33, DOI 10.3102/0013189X030002027, URL http://journals.sagepub.com/doi/10.3102/

0013189X030002027

Sadowski C, Zimmermann T (2019) Rethinking Productivity in Software Engineering. Apress Berkeley, CA

Saraste MK (2021) Playing as a Creative Tool for a Visual Artist: It's About Time-Teatime! Acrylic Painting as a Case Study. PhD thesis, Tampere University of Applied Sciences

Sarkar S, Parnin C (2017) Characterizing and Predicting Mental Fatigue during Programming Tasks. In: 2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion), IEEE, pp 32–37, URL https://doi.org/10.1109/SEmotion.2017.2

Schankin A, Berger A, Holt DV, Hofmeister JC, Riedel T, Beigl M (2018) Descriptive compound identifier names improve source code comprehension. In: Proceedings of the 26th Conference on Program Comprehension, ACM, Gothenburg Sweden, pp 31–40, DOI 10.1145/3196321.3196332, URL https://dl.acm.org/doi/10.1145/3196321.3196332

Schulte C, Clear T, Taherkhani A, Busjahn T, Paterson JH (2010) An introduction to program comprehension for computer science educators. In: Proceedings of the 2010 ITiCSE working group reports, ACM, Ankara Turkey, pp 65–86, DOI 10.1145/1971681.1971687, URL https://dl.acm.org/doi/10.1145/1971681.1971687

Scott JC (2016) Defeating the muse: Advanced songwriting pedagogy and creative block. In: The Routledge Research Companion to Popular Music Education, Routledge, num Pages: 13

Sennewald N (2021) Writer Types, Writing Strategies: Introducing a Non-English Text, Schreiben und Denken, to a New Audience. The Writing Center Journal 38(3):165–178, URL https://www.jstor.org/stable/27108280

Shaughnessy MP (1977) Errors and expectations: a guide for the teacher of basic writing. Oxford University Press

Siegmund J, Kästner C, Apel S, Parnin C, Bethmann A, Leich T, Saake G, Brechmann A (2014a) Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In: Proc. Int. Conf. Software Engineering (ICSE), ACM, pp 378–389, URL https://doi.org/10.1145/2568225.2568252

Siegmund J, Kästner C, Liebig J, Apel S, Hanenberg S (2014b) Measuring and Modeling Programming Experience. Empirical Software Engineering 19(5):1299–1334, URL https://doi.org/10.1007/s10664-013-9286-4

Siegmund J, Peitek N, Parnin C, Apel S, Hofmeister J, Kästner C, Begel A, Bethmann A, Brechmann A (2017) Measuring Neural Efficiency of Program Comprehension. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), ACM, pp 140–150, URL https://doi.org/10.1145/3106237.3106268

Software P (2021) Programmer's block: it happens to the best of us. URL https://www.parkersoftware.com/blog/programmers-block-it-happens-to-the-best-of-us/

Tompkins GE (1994) Teaching writing: Balancing Process and Product. Macmillan College

Videla A (2018) Lector in Codigo or The Role of the Reader. In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, ACM, pp 180–186, URL https://doi.org/10.1145/3191697.3214326

Whalley J, Kasto N (2014) A qualitative think-aloud study of novice programmers' code writing strategies. In: Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, Association for Computing Machinery, New York, NY, USA, ITiCSE '14, p 279–284, DOI 10.1145/2591708.2591762, URL https://doi.org/10.1145/2591708.2591762

Williams JD (2003) Preparing to teach writing: Research, theory, and practice. Routledge

Wymann C (2021) Mind Your Writing: How to be a Professional Academic Writer. Verlag Barbara Budrich, DOI 10.3224/84742459, URL https://library.oapen.org/handle/20.500.12657/60254

Züger M, Corley C, Meyer AN, Li B, Fritz T, Shepherd D, Augustine V, Francis P, Kraft N, Snipes W (2017) 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, pp 61–72, URL https://doi.org/10.1145/3025453.3025662