Three Lenses for Improving Programmer Productivity: *From Anecdote to Evidence*

Ph.D. Dissertation Proposal
Madeline Endres (endremad@umich.edu)

Overview: We propose a series of algorithms and theoretically-grounded interventions to enhance programmer productivity. By combining large-scale exploratory empirical investigations and controlled human-focused experimental design, we propose to both build mathematical models of the impact of understudied features on programmer productivity and to also provide actionable and evidence-backed interventions that improve productivity in practice for targeted diverse programmer groups. We propose three primary lenses: *developing efficient and usable bug-fixing tools* for non-traditional novices, *designing effective programming training* informed by objective measures of programming cognition, and *understanding the impact of external factors*, such as psychoactive substance use.

Intellectual Merit: In this proposal, we argue not only that varied external support can improve developer productivity, but we also specify which support can best do so. We contend that understudied factors and potential interventions can be identified through large-scale exploratory analyses. In addition, the impact of targeted interventions can be measured via causal experimental designs and large-scale human evaluations, even for factors impacting diverse populations that have previously only been considered anecdotally. We propose three research lenses into improving programmer productivity, each targeting a different type of support and programmer population.

1. **Developing Efficient and Usable Programming Tools:** We propose two novel methods of bug-fixing support targeting parse-errors and input-related bugs. Both are error types that we identify as commonly-encountered by *non-traditional novice programmers* (e.g., those learning without the support of the traditional classroom) but are overlooked by existing program-repair tools.
2. **Designing Effective Developer Training:** To help novice programmers become more like experts faster, we propose both developing a model of *novice programming expertise* using neuroimaging (fNIRS) and also leveraging our cognitive findings in the development and evaluation of a novel supplemental reading training for programming.
3. **Understanding External Productivity Barriers:** We argue that external factors also impact software productivity, including those anecdotally-reported but understudied by the scientific literature. We propose studying the impact of one such factor: *psychoactive substance use*. We propose both conducting the first survey of the prevalence of such substances in software and also developing a mathematical model of the true impact of one such substance, cannabis, on programming ability.

Broader Impact: Our proposed bug-fixing support for non-traditional novices could lower frustration faced by new programmers and thus lead to higher computing retention, especially for those without the support of the traditional classroom. Our proposed supplemental cognitive training could also broaden participation in computing; should our training transfer to improved programming outcomes, adding such training to computer science courses could decrease performance gaps for students with less incoming preparation in relevant cognitive skills, a preparation gap often correlated with factors including gender and socioeconomic status. Our proposed investigation of the intersection of psychoactive substances and programming, especially any findings related to hiring or retention, has the potential to influence drug-related policies in software. Outside of anecdote, little is known about the prevalence of such substances in software, let alone the accuracy of any perceived benefits. The proposed work has the potential to replace anecdote with evidence, enabling both companies and individual developers to make more informed decisions.

1 Introduction

Even as programming becomes increasingly integral to modern society, building software remains a challenging endeavor for both novice and expert programmers alike [54, 70, 85]. Understanding and supporting programmer productivity is key to the efficacy and efficiency of software development [80, 109]. Factors such as a lack of usable tool support [78, 113], inadequate training [1, 113], a negative non-technical environment [80], or even social biases in the workplace [37] can decrease developer productivity.

Many ways to improve developer productivity have been proposed and implemented (see Wagner and Ruhe for a review [117]). However, companies can struggle to identify what productivity factors and interventions best support their developers [80, 113]. For example, most proposed approaches focus on improving productivity through additional technical support, but the top productivity factors are often non-technical [80]. In addition, large productivity differences between software developers have long been observed [2, 99], even when controlling for common quantitative [99] or environmental [43] factors, suggesting that some developer populations are currently not well supported by existing techniques. Finally, companies may not be willing to take the risk to implement suggested productivity support (technical or not) until they know that it will work in practice: Adoption of new tools is low unless an explicit business case is presented [5] and trust is established with both managers and engineers [84].

This reluctance may in part be due to a history of suggested productivity support being overturned by later human-focused empirical evaluation. For example, in the automated debugging literature, a series of approaches provided developers with ranked lists of the possible code locations that were most likely to contain the error (see [55] as an indicative example). Once provided the faulty program line, the implicit assumption was that the developer could immediately identify and fix the issue, thus leading to increased productivity through faster bug finding and fixing. In practice, however, this turned out not to be the case: A human evaluation found that developers, when provided with this support, did not gain perfect bug understanding, instead skipping around the provided ranked list and wasting 61% of their time inspecting irrelevant statements [89]. Although this is but one example, we believe it to be indicative of a more general trend: that without considering the nuanced ways developers interact with their environment, even well-intentioned productivity interventions may fail to deliver their intended benefits in practice.

We propose a set of systematic studies that first identify previously-understudied but important factors that influence software productivity, and second provide rigorous human-focused evidence about the magnitude of their impacts, leading to actionable insights that generalize across programming populations. We argue that effective human-focused research on improving software productivity requires the following properties:

- **Provides Theoretically-Grounded and Actionable Insights:** We desire grounded mathematical models that capture practical aspects of developer productivity. Based on those models, we want to propose and evaluate specific interventions that can help programmers in practice.
- **Includes Empirical or Objective Measures:** Many existing approaches for studying developer productivity and well-being rely primarily on programmer self-reporting. While valuable, research from both Psychology and Software has found subjective self-reporting to be unreliable [26, 45]. When possible, we prefer augmenting self-reported data with more objective measures.
- **Minimizes Scientific Bias to Support Generalizability:** Studying humans can be challenging due to individual differences and sample bias. When possible, we favor leveraging techniques such as formal reasoning regarding theoretical outcomes, pre-registering hypotheses, and rigorous experimental control structures to minimize bias and promote generalizability.

- **Supports Diverse Developers:** Incoming preparation varies widely for software developers, leading to large productivity variance even between new developers who have the same undergraduate test scores or grades [99]. We want solutions that consider programmers from a diverse set of backgrounds, from non-traditional novices trying to learn programming outside of the classroom to those with different incoming cognitive abilities.

While previous research has considered factors that influence developer productivity and proposed developer support, it often lacks one or more of the listed elements. For example, one prominent line of work identifies and ranks the importance of various productivity factors, often using a combination of surveys and interviews [80, 109]. While helpful for prioritizing various factors for future study, such approaches both rely on subjective self-reporting [26] and also typically do not test the relations between such factors and productivity in a causal manner. Thus, this work does not itself propose specific testable interventions. We prefer approaches that use more objective human-focused measures, such as biometric data or behavioral programming logs. Another line of work seeks to improve productivity through proposing better developer tools including those that support debugging [65, 77]. While this work is important for advancing the state-of-the-art, most tool evaluations are entirely empirical, and do not explicitly consider the user (cf. [58]). Those studies that do evaluate their tool often assume unrealistic user properties or focus participant recruitment on a different population than the one the tool is designed for (e.g., students vs. professional developers), an oversight that can lead to tools not being useful in practice [40, 89]. We prefer solutions that explicitly consider human preferences via human studies with the target populations to include the likelihood of supporting diverse groups of programmers in practice.

We leverage three primary insights to conduct a set of studies that meet the aforementioned criteria of a good solution. **First**, we observe that exploratory empirical evaluations, especially those with hundreds of developers or millions of programming interactions, can identify previously-understudied productivity barriers in software for diverse sets of developers. This approach allows us to get an understanding of the anecdotal wisdom present in data such as developer-focused forums where programmers discuss concerns without oversight, often more likely to contribute negative experiences as well as positives. **Second**, we note that rigorous controlled experimental design can be used to discover evidence-backed conclusions to complex human-focused productivity questions. Real-world measurements of humans are inherently noisy. By carefully specifying the inputs and outputs in our experimental design, we will be better positioned to capture any true signals through the noise. **Third**, we argue that amelioration of productivity barriers actually encountered by developers in practice are more likely to be deployed or adopted. By providing generalized empirical evidence of the relevance of various factors and systematically validating the effectiveness of proposed interventions in diverse development contexts, we can offer solutions that are not only evidence-based but also tailored to the nuanced needs of specific programming communities.

We combine these insights into a set of systematic studies on understanding and improving programmer productivity. We consider three primary lenses: *designing efficient bug fixing algorithms* to help programmers quickly write more correct code, *developing effective developer training* to help novices become more like experts faster, and *understanding external productivity factors* to help all developers regardless of background succeed in computing and be happy while doing so. We propose research on specific instances of the lens (e.g., targeting a specific population or productivity factor). For each, we propose an initial exploratory phase and subsequent rigorous human evaluation and/or actionable solution. Each also combines interdisciplinary methodologies (including those from programming languages, psychology and medicine) with core software engineering techniques. Our specific proposed research components are as follows:

(1) Efficient Bug-fixing Support: Can we use programming languages techniques to support non-traditional novices in writing more correct code faster? Finding or fixing software bugs is one way

to improve software productivity [113]. Novice programmers can find debugging particularly challenging, especially those who do not have the support of the traditional classroom [11, 28]. We hypothesize that such non-traditional novices may not only struggle more with program errors in general, but also struggle with different errors than those faced by experts, error types not supported by existing tools. However, we argue that programming language techniques useful for supporting more expert programmers can be expanded to support non-traditional novices as well. In this research lens, we propose first identifying unsupported error types faced by non-traditional novices via empirical investigation, and second developing and evaluating support for two types of errors that we identified as barriers for novices: input-related bugs and parse errors.

(2) Effective Developer Training: Can we use medical imaging to inform supplementary cognitive training and improve programming outcomes? There is a growing body of work that uses neuroimaging to understand the cognitive processes behind programming (see [30] for a recent seminar). Such work, combined with a cognitive understanding of programming expertise, has the potential to lead to improved productivity through training targeted at identified relevant skills. However, this potential has yet to be explicitly tested in practice. We propose a two-phase approach: First, we will use neuroimaging to construct a mathematical model of novice programmer cognition. Second, we will combine insights from our model, and from other recent neuroimaging studies of programming, to develop a cognitive training curriculum likely to transfer to programming. We will evaluate the efficacy of our intervention using a controlled longitudinal study with programming students. By integrating cognitive insights with software engineering education, we aim to develop a novel approach to programmer training that is empirically grounded and directly applicable to the challenges faced by novices, especially those with less incoming preparation in programming-related cognitive skills.

(3) Understanding External Productivity Factors: How does psychoactive substance use impact software developer productivity? External factors beyond programming itself, such as cultural or environmental considerations, can also influence software developer productivity [43, 80]. Anecdotes connecting one such factor, psychoactive substance use, to programming abound. There are many conflicting opinions on substance-related cultural and productivity impacts on software (e.g., [8, 19, 74] for examples). However, little to no scientific research has been conducted to empirically assess these claims. We propose to fill this gap by first, conducting an exploratory survey on the relationship between psychoactive substance use and software. Second, we propose to conduct an observational study of the actual impact of one substance common in programming, cannabis, on programming productivity. We hope to help replace anecdote with evidence, enabling both companies and individual developers to make more informed and evidence-based drug-related decisions.

Overall, the thesis of this proposal is:

> *We can combine empirical evidence, theoretical modeling, and human-centered evaluation to develop and assess actionable interventions that improve programmer productivity in practice.*

Informally, we believe that more controlled methods from fields outside of software can improve software productivity for diverse groups. To the best of our knowledge, this proposal includes the first identification of, and support for, input-related bugs as a barrier for novices. We also propose the first controlled evaluation of the practical use of neuroimaging findings on software training as well as the first controlled study of psychoactive use and programming. The PI has published preliminary work supporting all three lenses, and has made publicly available relevant source code and data (when ethically permitted). To promote open and reproducible science, we will do the same for the remaining work, as well as pre-registering hypotheses for our proposed observational study of cannabis use and programming. In the rest of this proposal, we overview relevant background for each of the three lenses into developer productivity (Section 2),

describe the proposed research, including our primary insights (Section 3), outline our experimental design metrics and success criteria (Section 4), and present preliminary results (when available, Section 5).

2 Background

We present a high-level overview of background relevant to the research proposed in this proposal.

2.1 Automatic Program Repair

Automatic program repair (APR) is a set of techniques to automatically provide patches that fix buggy or incorrect programs. There is a vast literature on automatically repairing or patching programs: Proposed in 2008, APR has developed into a vibrant research area with over 500 publications (cf. the review [77]). Overall, traditional techniques take as input the source code for the buggy program and output a synthesized patch. Beyond program source code, APR techniques typically require as input the location of the bug (or some method for finding it), and some method to determine that a candidate patch correctly fixes the bug (e.g., a test suite or the like). The majority of approaches target expert programmers or professional software developers, and such expert-focused tools can be confusing for novices in practice [121].

APR has seen some adoption in industry, including use at Facebook (Meta) where it is used to repair bugs in multiple real-world systems at scale (e.g., each with millions of lines of code) [72]. In addition, the recent adoption of large language models have influenced APR approaches, with the use of models such as Codex [18] (which powered GitHub's Co-Pilot tool) or ChatGPT[1] becoming increasingly popular.

2.2 Program Comprehension and Neuroimaging

Software engineering researchers have long been interested in *program comprehension*, or understanding the cognitive processes that drive reading and writing source code [23]. By understanding what cognitive skills are relevant, researchers hope to gain insights that inform improved programming pedagogy and developer retraining, as well as to better understand productivity gaps between novices and experts [52]. Many cognitive skills have been associated with programming. Two of the more prominent ones are reading comprehension [15, 69, 79, 93] and spatial reasoning ability (a person's capacity to understand and reason about spatial relationships among objects, including activities such as mentally rotating 3D shapes). [73, 86, 88].

One way to study program comprehension is using *functional neuroimaging* to capture relevant brain activity. It can provide a physically-grounded insight into cognition without relying on potentially-unreliable self-reporting [30, 52]. We focus on one technique: *functional Near Infrared Spectroscopy* (fNIRS) which uses the *hemodynamic response*, or change in neuronal blood flow to active brain regions, to measure brain activity [16]. fNIRS can be used in ecologically-valid environments such as sitting at a standard desktop computer (cf. [63]). Following the pioneering work of Siegmund *et al.* [104], many works have used neuroimaging to investigate software, confirming the connections to both reading and spatial ability identified in the behavioral literature (e.g., [17, 27, 39, 42, 52, 53, 63, 81, 90, 104, 105]). Explicit studies of programming expertise are rare (cf. [91]), and tend to use proxies such as undergraduate grades or degree status.

2.3 Transfer Training

One potential intervention for supporting struggling novices is facilitating *learning transfer* to programming from a concurrently-taught supplemental training course [9], where learning transfer refers to the use of

[1] Produced by OpenAI, see `https://openai.com/chatgpt`

skills in a separate context from where they were learned. One of the most common cognitive training interventions proposed and validated for improving programming is spatial ability training (often in a one-credit class with one hour of additional instruction per week) [9, 21, 108, 114]. Broadly, supplemental spatial ability training can increase engineering degree retention [106] and directly improve programming outcomes [9, 21]. The impact on programming of training other relevant cognitive skills are less studied.

2.4 Cannabis and Its Use In Programming

Cannabis sativa is the world's most common illicit substance [110], used for both medical (e.g., pain, nausea) and recreational (e.g., social or perceptual enhancement, altered consciousness) purposes [6, 10, 66]. In the US, cannabis is criminalized, hampering research on its effects [82]. However, prohibition is contrary to popular opinion [57, 116], and 38 states have legalized cannabis for medical or adult use.

Anecdotes of cannabis use while programming abound. Questions inquiring about cannabis's effects on programming are common on online forums,[2][3] often inspiring numerous conflicting answers. Popular tech-related media sites cover the topic, positing that cannabis may help with programming-related chronic pain [59] or enhance programming through increasing focus [8] and creativity [118]. There is even evidence that cannabis use can impact software hiring and retention [19, 56, 67]. Physiologically, acute cannabis use impairs memory and learning as well inhibiting motor responses and reaction times [12, 62], cognitive processes that are used while programming [63, 104]. Cannabis may also impact creativity [60, 64], a key component of many software engineering tasks [47, 48, 71].

3 Proposed Research

We propose a series of experiments that both identify and try to ameliorate (either through a proposed intervention or evidence backed mathematical model) factors impacting developer productivity. These experiments are grouped into three different lenses on programmer productivity, each targeting a different sub-population or interdisciplinary methodology. We overview the proposed research for each lens:

- *Lens 1: Developing Efficient and Usable Programming Tools (for non-traditional novices)*
  - *1-a*: Factor Identification: We will identify error types that non-traditional novice programmers struggle with in practice, including those understudied in the existing literature.
  - *1-b*: Testable Solution: We will develop automated bug fixing tools supporting common error types, and establish this support's efficacy via both automated and human-focused evaluations.
- *Lens 2: Designing Effective Programmer Training (for students)*
  - *2-a*: Factor Identification: We will develop a mathematical model of cognitive processes important for new programmers.
  - *2-b*: Testable Solution: We will use existing cognitive understanding to design supplemental training. We will evaluate if our training transfers to programming via a controlled human study.
- *Lens 3: Understanding External Productivity Factors (for professional developers)*

[2] https://www.reddit.com/r/computerscience/comments/dbzp5v/how_many_of_you_found_that_smoking_weed_gets_you/

[3] https://news.ycombinator.com/item?id=509614

3-a: Factor Identification: We will conduct a survey of programmers on factors influencing software development productivity, including the prevalence and perception of psychoactive substances.

3-b: Testable Solution: We will conduct a controlled, observational study of one substance, cannabis, to develop a mathematical model of the impact of cannabis use on programming productivity.

We now discuss the research proposed in each lens in more detail.

3.1 Research Lens 1: Building Bug Fixing Support for Non-traditional Novices

Research Motivation and Overview. Novices are increasingly turning to online resources beyond the traditional classroom to learn computing [75,103]. But even as demand soars, the educational support provided by online tools leaves much room for improvement [11], especially for those students who need the most help [28]. Free tutoring environments seek to close this gap by providing educational support beyond structured course assignments. However, such sites can still suffer from low retention, reducing their ability to help in practice. We hypothesize that one reason for this low retention is the frustration novices face without instructional support; the time spent debugging a single error can correlate with student frustration [97]. In the proposed work, we hope to both identify and provide automated support for common errors encountered by non-traditional novices to help improve their productivity and lower computing barriers.

Factor Identification (*1-a*) — Where do non-traditional novices struggle? We first propose identifying error types that non-traditional novice programmers struggle with in practice via an empirical evaluation. To do so, we propose analyzing PYTHON TUTOR programming interactions; PYTHON TUTOR is a free online programming tutoring environment that is often used by novices learning programming without traditional classroom support [49, 50]. Two common error types encountered by novices merit particular attention: input-related errors and parse errors. For example, 35% of PYTHON TUTOR interactions involve user input, with 6.6% of interpreter errors fixed by only modifying input data. Similarly, 77.4% of all faulty programs failed with a parse error, accounting for the vast majority of the errors that novices face. Both of these error types are not currently well supported by Automatic Program Repair tools, which typically operate only on program source code (rather than inputs) and require the code to parse (see Section 2.1 for more detail).

Testable Solution (*1-b*) — Two novice-focused tools for bug-fixing support. While identifying factors non-traditional novices struggle with is important, we also wish to provide support for such errors. We propose two automatic bug fixing tools, one that targets program inputs rather than standard code and another that targets parse errors. We now discuss our proposed support for each type of error in more detail.

For input-related bugs, we propose INFIX, a randomized search optimization algorithm for automatically repairing program inputs for generic novice programs. The key insight behind INFIX is that input repairs are often composed from a small number of common mutations and that these mutations are often heavily correlated to specific error messages. As a result, we propose using an iterative approach where we repeatedly modify the buggy input according to error-message-specific mutation templates until it either finds a repaired input or times out. INFIX does not require test cases or training data to assess correctness. Instead, we propose using the implicit specification of eliminating interpreter errors (as tested by running the program after each input mutation). This, along with the fact that it permits a pleasingly parallel implementation, enables INFIX to provide efficient real-time programming support. To the best of our knowledge, we are the first to propose extending APR techniques to support input-related bugs (see [77] for a review).

For parse errors, we propose SEQ2PARSE, a language-agnostic approach that combines the theoretical accuracy of Error Correcting Parsers (EC-Parsers) [3] with the efficiency of Neural Networks to quickly generate high-quality fixes. In theory, EC-Parsers, which have been studied for decades but have not been widely adopted, can generate minimal-edit parse error repairs using special error production rules to handle

programs with syntax errors. Sadly, EC-Parsers (along with more modern equivalents such as [13, 22, 112, 115]) remain inefficient in practice, as their running time is cubic in the program size, and quadratic in the size of the language's grammar [76, 94]. Our key insight is that novice edits are highly predictable, and thus Neural approaches can pinpoint the relevant rules. Thus, we propose addressing parse error repair via a neurosymbolic approach where we first train sequence classifiers to predict the relevant EC-rules for a given program, and then use the predicted rules to synthesize repairs via EC-Parsing. To the best of our knowledge, we are the first to propose augmenting EC-Parsers with Neural methods.

For both tools, we propose evaluating a Python-supporting implementation on data from PYTHON TUTOR. For INFIX, we propose first developing Python-specific mutation templates via an observational study characterizing novice Python input patterns and errors. As both tools are aimed toward helping novices debug, we note that it essential that the repairs be helpful and of high quality. Previous work has shown that student-generated source repairs and expert human tutor input hints can be helpful for students by decreasing debugging time and increasing learning. Therefore, we propose augmenting automatic evaluation with human studies comparing tool-made repairs to those developed by the PYTHON TUTOR users themselves.

3.2 Research Lens 2: Designing Effective Programmer Training via Cognitive Insights

Research Motivation and Overview. As discussed in Section 2.2, the software engineering community has increasingly used medical neuroimaging to understand the cognitive processes behind programming. Such studies have the potential to improve our understanding of expertise and to inform software engineering pedagogy, helping novices become experts faster (see Floyd *et al.* [42, Sec. II-D] for a summary). Many of the neuroimaging studies in software engineering have compared programming to reading [42, 104, 105] or spatial manipulation [52]. Tantalizingly, one study [42] found that coding became more neurologically similar to reading for programmers with even greater expertise.

Critically, most (c.f., [91]) software engineering neuroimaging studies thus far have only studied programming *experts* that are either professionals or students with multiple years of experience (e.g., [105, Sec. 3.3]). In addition, the potential of neuroimaging to improve pedagogy has not yet been tested in practice. To realize this potential, we propose both directly observing true novices with neuroimiging and also studying the potential pedagogical impact via the design and evaluation of cognitive supplemental training.

Factor Identification (*2-a*) — A mathematical model of novice programmer cognition. To better understand the cognitive processes of novice programmers, we propose using fNIRS to conduct a controlled neuroimaging study with first-semester programming students. We propose comparing participants' brain activation patterns while coding to those while reading prose or using spatial reasoning (i.e., mentally rotating 3D objects, see Section 2.2) to see if patterns observed with more expert developers continue.

We note that studying novices presents challenges relative to studying experts. First, we must create experimental stimuli amenable for novices with little to no previous coding experience; the coding stimuli in previous neuroimaging studies all involve constructs (e.g., trees [52]) or tasks (e.g., code review [42]) unfamiliar to novices. We propose leveraging introductory computing syllabi to develop stimuli commensurate with participant experience. Second, we must pay particular care to recruit participants with equivalent programming expertise; even though we are recruiting from the same introductory course, programming experiences can vary [119]. To mitigate this risk, we propose restricting our population to students without previous programming experience. Finally, we propose a protocol that follows up with participants months later to assess their programming gains using a written assessment. This provides a preliminary exploration of an aspect of learning, assessing if novices' brain activation patterns while coding can predict their future programming ability and underscoring the potential for cognitive interventions in computing.

Testable Solution (*2-b*) — Does technical reading training transfer to programming outcomes? As discussed in Section 2.3, one way neuroimaging may impact practice is by informing targeted cognitive training that *transfers* to improved programming outcomes. Such transfer for spatial reasoning has been observed in the educational literature [9, 21, 108, 114]. However, the benefit of training of other cognitive skills identified by neuroimaging is less understood. We hypothesize that reading ability may sometimes be more important than spatial ability for software engineering success: Many essential software engineering tasks, such as code review, code summarization, or documentation, require technical reading ability.

To test this hypothesis, we propose a semester-long CS-focused reading training course that covers various technical reading strategies. The majority of these strategies will focus on using structural cues to quickly and accurately scan texts to retrieve and understand key points. Topics will include focusing on outlines when reading to improve comprehension, understanding figures and charts in scientific writing, and strategies for understanding persuasive technical proposals (e.g., Heilmeier's Catechism). Our emphasis on teaching structural cues and patterns is motivated by findings that experienced programmers tend to read code non-linearly, focusing on high level features [14, 96]. We hypothesize that explicitly teaching how to trace through scientific texts will transfer to tracing through programs, a key software activity.

We propose evaluating our technical reading course via an 11-week randomized trial where we compare our novel technical reading training and a validated standardized spatial ability training, each with the same student time commitment of 1.5 hours per week. We propose doing so because first, spatial training has already been found to transfer to programming, so it can act as a strong control and second, comparing the two interventions allows us to come to actionable conclusions about which intervention is most helpful for introductory computing. To the best of our knowledge, we are the first both to propose a computing-focused technical reading training and also to test for transfer to computer science.

3.3 Research Lens 3: External Productivity Factors: A Cannabis Case Study

Research Motivation and Overview. For the final research lens we investigate how external factors can impact programmer productivity and well-being. There are a large array of potential external factors that could influence programming. We propose focusing on psychoactive substances with a particular focus on cannabis use: Despite anecdotal connections and potential impacts (see Section 2.4), no previous empirical studies have directly investigated the intersection of software and psychoactive substances. Nor have any observational studies investigated the actual effects of such substances, including cannabis, on programming ability. We hope to fill this gap with evidence-based research that either confirms or overturns conventional wisdom, helping programmers and policy makers make more informed decisions.

Factor Identification (*3-a*) — A survey of psychoactive substance use in software. To gain an empirical understanding of the prevalence and perceptions of psychoactive substances in software, we propose conducting a survey of programmers. As we would like our survey results to be useful to company policy-makers, it is essential that our population includes professional developers. However, as many psychoactive substances are illegal or explicitly prohibited by corporate policy (e.g.,see [20] for an example), it is also imperative to maintain participant privacy and confidentiality to avoid the risk of work place retaliation. To balance these considerations, we propose focusing recruitment on top contributors to open source projects (who are often professionals [95]) rather than on software engineering company contacts.

Testable Solution (*3-b*) — How does cannabis actually impact programming? We propose an observational study to develop a mathematical model of the actual effects of one substance, cannabis, on developer productivity. As discussed in Section 2.4, it seems likely that many programmers use cannabis while programming, holding positive views of its impacts. In broader contexts, however, views may be more negative.

Cannabis can impair decision-making consistency, motor control, and reaction times [62, 98], impacts that inform both general and software-specific regulation. However, the efficacy of such policies [83, Sec. IX.B] and whether they are needed or beneficial [56] have been questioned in a modern context.

We hope our proposed study will fill this knowledge gap, allowing for evidence-based corporate policies and informed decisions by developers regarding when and if to use cannabis while programming. We propose a study that first has a larger, indicative sample size (e.g., more than a dozen participants, more than just students), second uses ecologically valid conditions (e.g., tasks that are used in the real world, cannabis dosages consistent with actual developer use), and third captures both quantitative (e.g., number of defects or typing speed) and qualitative (e.g., creativity, coding style) aspects of produced software. Specifically, we propose using a counter-balanced within-subjects design to maximize statistical power when comparing programming outcomes while intoxicated (at the level they would normally use while programming) to those while sober. We propose holding sessions remotely with the participant using their own computer to both ensure participant comfort and safety while also maximizing experimental ecological validity.

4 Proposed Research Questions and Metrics

We discuss our experimental design, metrics, and success criteria for evaluating each proposed research component in more detail. We will correct for multiple comparisons, as well as compute effect sizes and statistical significance as appropriate throughout our mathematical analyses across all research questions.

4.1 Experimental Design — Lens 1: Bug Fixing Support for Non-Traditional Novices

When investigating how to support non-traditional novices to fix software bugs (see Section 3.1), we consider three research questions: 1) *RQ1 — Factor Identification:* What types of errors do non-traditional novices struggle with the most? 2) *RQ2-a — Testable Solution:* Are our proposed tools, INFIX and SEQ2PARSE, able to accurately and efficiently repair input-related bugs and parse errors respectively, as compared to state of the art approaches? 3) *RQ2-b — Testable Solution:* Are the repairs produced by INFIX and SEQ2PARSE of high quality compared to historical user repairs, as judged by humans?

RQ1: To understand how non-traditional novices struggle with bug fixing, we propose an empirical analysis of four years of data (including over six-million programming interactions) from PYTHON TUTOR, a free online programming tutoring platform [50]. *Metrics:* We will sort the most common error types in the data set using both frequency (number of occurrences) and difficulty (average time for the student to fix). We will consider error types to be of high-priority if they are either of high frequency (account for more than 5% of all errors) or are difficult (often take more than two minuets to solve).

RQ2-a: To understand the accuracy and efficiency of INFIX and SEQ2PARSE, we will evaluate our implementations using relevant subsets of the PYTHON TUTOR dataset. For SEQ2PARSE, we will train and test on separate subsets. *Metrics*: For both tools, our primary metric will be *error repair rate*. That is, for what percentage of buggy scenarios can our tool fix the program such that it parses or runs without error? For SEQ2PARSE, we will also consider what fraction of repairs exactly match the historical human repair. For both, we will measure efficiency by calculating the average time to repair. We will consider our tools accurate if they have repair rates commensurate with state-of-the-art APR tools such as DEEPFIX, which repairs 27% of programs [51], and GGF, which repairs 58% of syntax errors [120]. We will consider our tools efficient if they can produce the majority of their repairs in real time (i.e., less than six seconds).

RQ3-b: To understand how our repairs compare to human-generated repairs, we will conduct a human study. We will recruit participants via mailing lists, professional contacts, and social media. Participants

will be directed to an online web portal where they will be asked to rate the quality and helpfulness of a series of program repairs using a Likert scale. Stimuli will be selected randomly from the PYTHON TUTOR dataset, and each will have two versions: the tool-generated repair and the historical student repair. *Metric*: We will compare Likert scores between the tool-generated and human-generated versions to see if there is a statistical difference in quality. We will consider repairs to be of high quality if they are judged more similarly to historical human repair than repairs from other APR tools (i.e., above 75%, see [58, Tab. VII]).

4.2 Experimental Design – Lens 2: Effective Programmer Training via Cognitive Insights

We organize our investigation of cognitive modeling and transfer training (see Section 3.2) around three research questions. The first two relate to the fNIRS study, while the third involves the proposed reading training: 1) *RQ1-a — Factor Identification:* What areas of the brain activate when novice software engineers program, and how do they compare to brain activation during mental rotation or reading? 2) *RQ1-b — Factor Identification:* Can brain activation patterns at the beginning of CS1 predict programming performance at the end of the course? 3) *RQ2 — Testable Solution:* Does our reading training transfer to improved programming outcomes, and if so, is this effect more pronounced for some programming aspects than others?

RQ1-a and RQ1-b: As described in Section 3.2, we propose using fNIRS to study novice programmers completing spatial ability, reading, and programming problems. We will present these stimuli in a randomized order using a block-based design, allowing us to *contrast* brain activity between tasks. For spatial ability, we propose using mental rotation stimuli adapted from Peters and Battista's Mental Rotation Stimulus Library [92]. For reading, we propose using sentence completion tasks adapted from official *Graduate Record Examination* practice questions [38]. For programming stimuli, we propose creating a corpus of short code snippets that use constructs familiar to introductory computing students such as Boolean logic, loops, and arrays [111]. For the programming assessment at the end of the course, we propose using the Second CS1 Assessment (SCS1) [87], a validated language-agnostic measure of CS1 programming ability.

Metrics: To analyze the fNIRS data, we will follow established best practices: We will model the hemodynamic response for each subject using a General Linear model. We will then use a Linear Mixed Effects model for group level analysis and compute t-value and p-value statistics for brain activation contrasts between tasks. Results will be significant if $p < 0.01$ with a false discovery threshold $q < 0.05$. For analyzing if brain activation patterns can predict programming outcomes, we propose using *Representational Similarity Analysis* (RSA), a common Psychology approach [61]. We will reject the null hypothesis if we retain a significant correlation with absolute value ($r > 30$, $p < 0.01$) after multiple comparison correction.

RQ2: To evaluate our semester-long reading training, we will compare the programming gains between participants in the reading training to those from a validated spatial training course developed by Sorby *et al.* [107, 108]. Due to its established correlation with computer science outcomes and evidence of transfer to programming ability [9, 106], we propose using spatial training as a baseline for analyzing the efficacy of our reading treatment. To ensure a fair comparison, both courses will have the same weekly time commitment and participants (all recruited from the same course) will be randomly assigned to an intervention.

Metrics: We will assess programming ability gains between groups using a pre-test/post-test design with the SCS1 [87]. The SCS1 has three types of questions: definition, code tracing, and code-completion. To see if our reading training transfers to programming, we will specify a multiple linear regression model containing pre-test programming scores (to control for participant variation) and intervention group as predictors. To see if the effect is different for different question types, we will specify a multiple linear regression model for each with sub-type pretest scores and intervention group as predictors. We will consider the difference in programming gains between the two groups significant if $p < 0.05$.

4.3 Experimental Design — Lens 3: External Factors, Cannabis and Programming

In our third lens we propose using a large-scale survey followed by an observational study to understand the impact of cannabis use on programming. We consider two primary research questions: 1) *RQ1 — Factor Identification:* Do programmers use psychoactive substances, such as cannabis, while programming? If so, when and why do they do so? and 2) *RQ2 — Testable Solution:* Does cannabis intoxication while programming impact program correctness, programming speed, or programming creativity?

RQ1: To capture the experiences of professionals and students alike, we propose a survey recruiting from popular open-source GitHub projects and current computing students at the University of Michigan. To measure *cannabis usage frequency*, we propose using the validated Daily Sessions, Frequency, Age of Onset, and Quantity of Cannabis Use Inventory (DFAQ-CU) [25]. To measure cannabis use *while programming*, we propose adapting questions from the DFAQ-CU by adding the phrase "while programming, coding, or completing any other software engineering-related task?". We will also ask participants for which types of programming projects or tasks they are likely to use cannabis, and about their usage motivations. Choices will reflect those we observed in anecdotal online posts, and include a free text option. *Metrics:* We will compare variables (e.g., Likert perception scores, participant age, etc.) using t-tests, χ^2-tests, and proportions z-tests as appropriate. $p < 0.05$ will indicate significance, and within each research question, we will correct for multiple comparisons using a Benjamini-Hochberg Threshold of $q = 0.05$.

RQ2: We propose conducting an observational study to develop a model of the impact of cannabis use on programming. We propose using a within-subjects design where each participant completes two sessions on different days, one while sober and one while using cannabis, in a counterbalanced order. In each session, participants will complete both short programming tasks from the neuroimaging literature [63] and also multiple LeetCode problems [7] using a standard development environment (Visual Studio) on their personal computers. In the cannabis condition, participants will use the amount they would normally use while programming, an ecologically-valid context that allows us to learn actionable insights. Based on our power analysis where we estimate our expected effect size with the known effect of cannabis on programming-related cognitive tasks such as working memory, we propose recruiting at least 50 participants

Metrics: We propose capturing participant key strokes, program runs, and completion time. We will measure cannabis's impact on program correctness by comparing scores on held out test suites that we design via common systematic processes (i.e., achieve full branch coverage [46]). We will compare programming speed using the average number of key strokes per second and the total time to complete the problem. We propose to assess one aspect of programming creativity via a qualitative approach where we manually annotate responses for style and algorithmic choices that may differentiate high and sober programmers.

5 Preliminary Results

We now cover available preliminary results for each lens.

5.1 Preliminary Results – Lens 1: Building and Evaluating Bug Fixing Support

Preliminary work on INFIX and input-related bugs was published in the International Conference on Automated Software Engineering, 2019 [34], while SEQ2PARSE and parse errors was published in the OOPSLA issue of the Proceedings of the ACM on Programming Language in 2022 [101]. The human-evaluations of both tools were approved by Michigan's IRB under HUM00158717 and HUM00183855.

RQ1 – Factor Identification, Common Novice Errors: We find that the most common error type encountered by non-traditional novices is parse-errors: 77.4% of all faulty programs in the PYTHON TUTOR dataset failed with a syntax error. Beyond being ubiquitous, parse errors can take time for novices to fix: 37% take over two minutes to resolve, with more complex fixes taking longer. Even for fixes that require only a one or two token change, the user spends 25 seconds on average (a considerable amount of time for such simple fixes) — this time jumps to 56 seconds for three token fixes. Together, we observe strong indications that parse errors are a very common category of error for which novices may benefit from additional support.

We find that input-related bugs are also common and time-consuming for novices. Making up 6.6% of all PYTHON TUTOR errors, programs with input-related errors are often surprisingly complex. 40% of erroneously-used input calls are embedded in either a loop or a `split` call, resulting in complex dynamic input interaction. As a result, input-related errors take a wide range of time for users to resolve. The median time to solve input-related errors is 49 seconds, however 34.6% take users over two minutes to solve and 5.5% take over seven minutes. Based on the metrics proposed in Section 4.1, we find that both parse errors and input-related bugs are common and challenging, making them high priority for providing additional support. This helps us gain confidence in our over-arching insight that large empirical evaluations can identify previously-understudied programming barriers.

RQ2-a — Testable Solution, Automated Evaluation: We find that our proposed support for input-related errors, INFIX is able to both accurately and efficiently repair historical student bugs, repairing 96% of input-related errors in the PYTHON TUTOR dataset in a median of 0.88 seconds. SEQ2PARSE is similarly accurate and efficient, able to parse and repair 94% of syntax errors in a median of 2.1 seconds. These numbers demonstrate that both tools are accurate and efficient in generating repairs: Both have a repair rate well above those of state-of-the-art APR tools and are also efficient enough to provide real-time feedback to struggling learners (see Section 4.1 for more). Our results demonstrate the potential of exploratory empirical evaluations to lead to actionable insights and tools for supporting programmers from diverse populations.

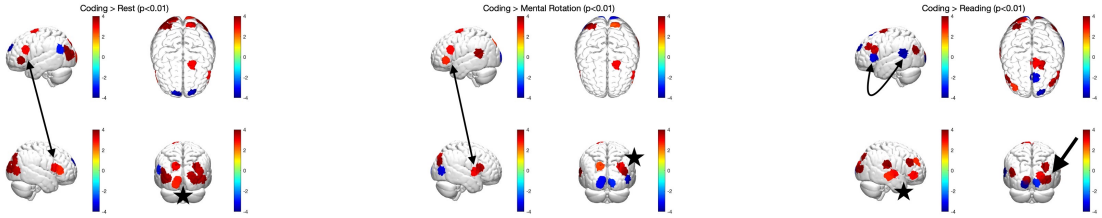
RQ2-b — Testable Solution, Human Evaluation: In our preliminary work, we find that repairs produced by both INFIX and SEQ2PARSE are well-received in comparison to the historical repair. In human studies with 97 and 48 participants, we found that INFIX and SEQ2PARSE repairs (respectively) were similar in quality (and sometimes of higher quality) than the historical bug fix. For example, participants found INFIX's repairs equally helpful as human repairs and we found that 35% of SEQ2PARSE repairs are actually equivalent to the historical fix. Of those that remain, 15% were judged to be significantly more useful and 52% equally useful: k. Our preliminary results demonstrate how tools designed to support commonly-encountered bugs can be well-received in practice, helping programmers write more correct code faster.

5.2 Preliminary Results — Lens 2: Effective Programmer Training via Cognitive Insights

Preliminary results for our fNIRS study on programming expertise an the efficacy of our reading training were published in the 2021 International Conference on Software Engineering [32], and the 2021 Symposium on the Foundations of Software Engineering [31], respectively. Both phases of the proposed research involve human subjects, which was approved by Michigan's IRB under HUM00173556.

RQ1 — Factor Identification, Programming Brain Activation: To develop a preliminary mathematical model of novice programmer cognition, we conducted a study with 31 first-year computing students using fNIRS (see Section 2.2). We captured brain activation during programming, reading, and spatial reasoning tasks, comparing observed activation to both a rest condition and also to each other.

Figure 1a shows our preliminary results for Programming vs. Rest; we found that novice programmers engage brain regions associated with language and spatial cognition, as well as regions associated with in-



(a) Programming vs. Rest (b) Programming vs. Rotation (c) Programming vs. Reading

Figure 1: Significant Brain Activation Contrasts for Novice Programmers. Red indicates regions more activated during coding, while blue indicates regions more activated during the contrast activity. Arrows and stars indicate brain activation that is particularly distinctive between the two tasks.

creased demand for attention and executive function ($p < 0.01$, $q < 0.05$). In addition, as demonstrated by the contrasting brain activation in Figures 1b and 1c, for novices, coding is neurally distinct from both reading and spatial reasoning ($p < 0.01$, $q < 0.05$). Coding engages regions associated with working memory (the cognitive system used to store and process information over short periods) more than does either reading or rotation, indicating that programming is more cognitively challenging. However, unlike previous work with experts that found strong similarities between coding and reading [42,104,105], we observe more substantial differences between coding and reading than we do for coding and spatial tasks. This indicates that novices rely heavily on visiospatial cognitive processes while coding. Overall, our preliminary results demonstrate the ability of objective methods such as fNIRS to identify programming-relevant cognitive processes and help develop a cognitive model of computing that that is more objective than self-reporting.

RQ1-b — Factor Identification, Predictive Brain Activity: In our preliminary results, we found that brain activation patterns captured at the beginning of a semester can indeed predict end-of-semester programming outcomes. We found a significant medium negative correlation between the similarity of a participant's mental rotation and coding brain activation patterns and their final programming score (in the right frontal region of the brain, $r = -0.48$, $p = 0.006$): the less similar the neural activation patterns for coding and rotation, the better the final programming assessment outcome.

While more work is needed to fully understand the mechanisms facilitating this finding, it may indicate that novices who use more problem-solving intensive strategies at the beginning of the semester make less progress over in a semester. This hypothesis may align with the Spatial Encoding Strategy framework, a leading theory of spatial encoding in computing which hypothesizes that strong spatial reasoning ability helps novice programmers with general, but not with domain-specific, strategies for mentally encoding programming information [73]. Our work may thus have implications for the use of domain-specific strategies in skills-based training. Regardless, our preliminary findings demonstrate not only the ability of measures such as fNIRS to gain computing insights more objective than self-reporting [45], but also that they can identify factors that have real-world predictive power (e.g., could be leveraged in interventions that help improve productivity in practice).

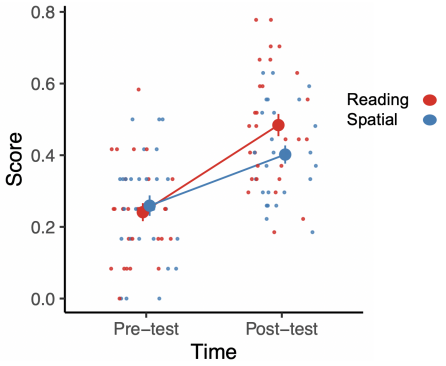


Figure 2: Reading participants have significantly larger % programming gains.

RQ2-a — Testable Solution, Supplemental Reading Training. Having established that spatial cognition

is relevant for novice programmers, we also wanted to see if cognitively-informed supplemental reading training (see Section 3.2) is as effective. To test for transfer from our training to improved programming, we conducted a controlled 11-week longitudinal study with 57 CS1 students. We compared the programming-related impacts of our novel reading training curriculum and a validated spatial training curriculum by comparing final programming scores on the (SCS1 [87], see Section 4.2).

We find our reading training improves programming outcomes: as shown in Figure 2, students in our reading training performed significantly better on the final programming test than those in the spatial training (via a multiple linear regression with pre-test programming scores and training group as predictors, $B = -0.09$, $SE(B) = 0.14$, $t(54) = -2.33$, $p = .02$). This indicates that technical reading ability may sometimes better facilitate programming for novices than spatial ability, and demonstrates the importance of objective methods (such as fNIRS) for informing actionable interventions.

We found that the programming-related benefits of our reading training is largest for code-tracing questions ($B = -1.04, SE(B) = 0.45, t(54) = -2.31, p = .03$). This demonstrates the potential usefulness of our curriculum in practice: for novices, the ability to trace through code and describe its function in natural language is highly-predictive of programming ability [69, 79]. Code tracing remains essential for experienced programmers as developers are frequently required to read and understand code to both contribute to multi-programmer projects [44] and also for code reviews [4]. Overall, our preliminary results demonstrate the potential of cognitive training to support the productivity of novice programmers, especially those who may have lower incoming preparation in cognitive skills such as technical reading or spatial ability.

5.3 Preliminary Results — Lens 3: Psychoactive Substances and Programming

We have preliminary results for this lens's first research question (see Section 4.3). We conducted a survey of the psychoactive substance usage history and preferences of 803 programmers, including 450 full-time developers. This survey was supplemented by hour-long interviews with 26 professional programmers who use psychoactive substances while programming. Both the survey and subsequent interviews were approved by the IRB (HUM00187787, and HUM00213745), and preliminary results were published in the International Conference on Software Engineering in 2022 [29] and 2023 [83].

RQ1 — Factor Identification, Psychoactive Substance Usage Patterns: We find that psychoactive substance use is common in software: in the last year, 59% of our respondents have used a mind-altering substance while programming, with alcohol and cannabis the most common. For example, we find 35% of our sample have ever used cannabis while completing a programming or software engineering-related task, of which half currently use cannabis at least once or more a month. This use is most commonly-motivated by perceived programming enhancement or increased enjoyment. In addition, we find evidence that strict anti-drug-policies can influence job search decisions, with some developers reporting that the existence of such a policy reflects poorly on company culture and developer trust.

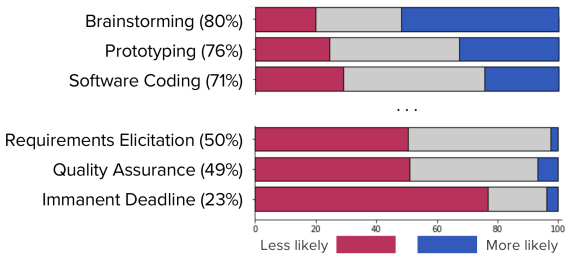


Figure 3: Selected tasks that are more or less likely completed while using cannabis.

Regarding cannabis in particular, we find that programmers often use products that contain THC (the compound responsible for its mind-altering effects), and 11% of those who have used while programming report currently doing so on a near daily basis, behaviors very likely to be detected by most drug test policies. This is especially relevant as we find that over

a third of cannabis-using programmers sometimes choose to use during work-related tasks, a likelihood that increases for developers working remotely. As depicted in Figure 3, programmers may self-regulate this work-related use by software task: cannabis use is more likely during creative open-ended software tasks (such as Brainstorming or Prototyping) than during time- or safety- critical tasks (such as Quality Assurance), a finding that may have policy implications.

Our preliminary findings of frequent use by professional programmers and reports of perceived programming enhancement further motivate our proposed and ongoing observational study of the actual effects of cannabis on programming. We have received IRB approval for a controlled observational study of cannabis's impact on programming (HUM00223584). We have secured funding for participants, and have successfully obtained preliminary data from 74 participants. Our preliminary results give us confidence in the potential of the proposed work to provide actionable insights for evidence-based drug policy reform in software.

6 Schedule

The proposed work for lens one was started in 2018 and completed in 2022 (with supporting publications in 2019 and 2022). The proposed work for lens two was started in 2019 and completed in 2021 (with supporting publications in 2021). The proposed work for lens three was started in 2021. The preliminary results for the first research question were finished in 2023 (with supporting publications in 2022 and 2023). The proposed work for lens three's Testable Solution is expected to be finished and published no later than February 2024. We propose targeting the International Conference on Software Engineering for the remaining work.

7 Conclusion

Improving programming productivity, for both novices and experts, has long been a key challenge in software. Myriad potential relevant factors and interventions have been proposed. However, many have failed to impact practice, in part due to limited human-focused evaluation, causing companies or educators to struggle to identify and implement proposals. We argue that effective research on programming productivity should provide theoretically-grounded and actionable insights, include empirical or objective measures to validate self-reported data, and support the diverse backgrounds of developers. We suggest that a combination of exploratory empirical evaluations, rigorous experimental design, and a focus on practical productivity barriers can yield actionable results. These insights guide our approach to identifying understudied productivity factors and evaluating their impact on a broad spectrum of developers.

We propose three targeted research initiatives. First, we aim to develop algorithms that support bug-fixing for non-traditional novices, focusing on common error types such as input-related bugs and parse errors that we find such novices encounter in practice. Second, we suggest improving productivity via evidence-backed developer training; we propose using neuroimaging to both build a mathematical model of programming expertise and also inform cognitive training that helps novice programmers become experts faster. Third, we propose understanding one way that external factors can impact software via an exploration of the impact of psychoactive substances on programming productivity, a subject full of anecdotal evidence but lacking sufficient scientific scrutiny. The PI is well positioned to carry out the research in all three lenses, with 14 peer-reviewed conference and workshop publications [24, 29, 31–36, 41, 68, 83, 100–102], including six [29, 31, 32, 34, 83, 101] that directly support the proposed work. By addressing these lenses into programming productivity through empirical investigation and rigorous human evaluation, we aim to contribute to the creation of a more productive and inclusive software development environment.

References

[1] T. Acton and W. Golden. Training the knowledge worker: a descriptive study of training practices in irish software companies. *Journal of European industrial training*, 27(2/3/4):137–146, 2003.

[2] H. Ahmad, Z. Karas, K. Diaz, A. Kamil, J.-B. Jeannin, and W. Weimer. How do we read formal claims? eye-tracking and the cognition of proofs about algorithms. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 208–220. IEEE, 2023.

[3] A. V. Aho and T. G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4):305–312, 1972.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[5] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings 4*, pages 1–20. Springer, 2004.

[6] E. P. Baron, P. Lucas, J. Eades, and O. Hogue. Patterns of medicinal cannabis use, strain analysis, and substitution effect among patients with migraine, headache, arthritis, and chronic pain in a medicinal cannabis cohort. *The journal of headache and pain*, 19(1):1–28, 2018.

[7] M. Behroozi, C. Parnin, and T. Barik. Hiring is broken: What do developers say about technical interviews? In J. Smith, C. Bogart, J. Good, and S. D. Fleming, editors, *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*, pages 1–9. IEEE Computer Society, 2019.

[8] M. Berman. How cbd oil can help programmers focus. `https://programminginsider.com/how-cbd-oil-can-help-programmers-focus/`, 1 2020. Accessed: 2021-03-07.

[9] R. Bockmon, S. Cooper, W. Koperski, J. Gratch, S. Sorby, and M. Dorodchi. A cs1 spatial skills intervention and the impact on introductory programming abilities. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 766–772, 2020.

[10] K. F. Boehnke, S. Gangopadhyay, D. J. Clauw, and R. L. Haffajee. Qualifying conditions of medical cannabis license holders in the united states. *Health Affairs*, 38(2):295–302, 2019.

[11] C. J. Bonk, M. M. Lee, X. Kou, S. Xu, and F. Sheu. Understanding the self-directed online learning preferences, goals, achievements, and challenges of MIT opencourseware subscribers. *Educational Technology & Society*, 18(2):349–365, 2015.

[12] S. J. Broyd, H. H. van Hell, C. Beale, M. Yücel, and N. Solowij. Acute and chronic effects of cannabinoids on human cognition—a systematic review. *Biological Psychiatry*, 79(7):557–567, 2016. Cannabinoids and Psychotic Disorders.

[13] M. G. Burke and G. A. Fisher. A practical method for LR and LL syntactic error diagnosis. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, 1987.

[14] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.

[15] T. Busjahn, C. Schulte, and A. Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9, 2011.

[16] R. B. Buxton, K. Uludağ, D. J. Dubowitz, and T. T. Liu. Modeling the hemodynamic response to brain activation. *Neuroimage*, 23:S220–S233, 2004.

[17] J. Castelhano, I. C. Duarte, C. Ferreira, J. Duraes, H. Madeira, and M. Castelo-Branco. The role of the insula in intuitive expert bug detection in computer code: an fmri study. *Brain imaging and behavior*, 13(3):623–637, 2019.

[18] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[19] M. Cherney. The fbi says it can't find hackers to hire because they all smoke pot. `https://www.vice.com/en/article/d737mx/the-fbi-cant-find-hackers-that-dont-smoke-pot`, 5 2014. Accessed: 2021-03-07.

[20] Cisco. 2019 code of business conduct. `https://www.cisco.com/c/dam/en\\\\_us/about/cobc/2019/english-2019.pdf`, 2019. Accessed: 2021-08-09.

[21] S. Cooper, K. Wang, M. Israni, and S. Sorby. Spatial skills training in introductory computing. In *International Computing Education Research*, pages 13–20, 2015.

[22] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro. Repairing syntax errors in lr parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):698–710, 2002.

[23] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Software Eng.*, 35(5):684–702, 2009.

[24] B. Cosman, M. Endres, G. Sakkas, L. Medvinsky, Y.-Y. Yang, R. Jhala, K. Chaudhuri, and W. Weimer. Pablo: Helping novices debug python code through data-driven fault localization. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1047–1053, 2020.

[25] C. Cuttler and A. Spradlin. Measuring cannabis consumption: psychometric properties of the daily sessions, frequency, age of onset, and quantity of cannabis use inventory (dfaq-cu). *PLoS One*, 12(5):e0178194, 2017.

[26] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies. " yours is better!" participant response bias in hci. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 1321–1330, 2012.

[27] J. Duraes, H. Madeira, J. Castelhano, C. Duarte, and M. C. Branco. Wap: understanding the brain at software debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 87–92. IEEE, 2016.

[28] S. Dynarski. Online courses are harming the students who need the most help. In `https://www.nytimes.com/2018/01/19/business/online-courses-are-harming-the-students-who-need-the-most-help.html`, 2018.

[29] M. Endres, K. Boehnke, and W. Weimer. Hashing it out: A survey of programmers' cannabis usage, perception, and motivation. In *International Conference on Software Engineering*, page 1107–1119, 2022.

[30] M. Endres, A. Brechmann, B. Sharif, W. Weimer, and J. Siegmund. Foundations for a new perspective of understanding programming (dagstuhl seminar 22402). *Dagstuhl Reports*, 12(10):61–83, 2022.

[31] M. Endres, M. Fansher, P. Shah, and W. Weimer. To read or to rotate? comparing the effects of technical reading training and spatial skills training on novice programming ability. In *Foundations of Software Engineering*, pages 754–766, 2021.

[32] M. Endres, Z. Karas, X. Hu, I. Kovelman, and W. Weimer. Relating reading, visualization, and coding for new programmers: A neuroimaging study. In *International Conference on Software Engineering*, pages 600–612, 2021.

[33] M. Endres, P. Reiter, S. Forrest, and W. Weimer. What can program repair learn from code review? In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 33–34, 2022.

[34] M. Endres, G. Sakkas, B. Cosman, R. Jhala, and W. Weimer. Infix: Automatically repairing novice program inputs. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 399–410. IEEE, 2019.

[35] M. Endres, W. Weimer, and A. Kamil. An analysis of iterative and recursive problem performance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 321–327, 2021.

[36] M. Endres, W. Weimer, and A. Kamil. Making a gamble: Recruiting se participants on a budget. In *1st Workshop on Recruiting Participants for Empirical Software Engineering*, 2022.

[37] K. N. B. Enriquez, A. M. S. Hidalgo, R. F. T. Quina, N. J. L. Valencia, and J. R. M. Buzon. The effect of gender inequality on job satisfaction, productivity, and career progression of female it and software professionals. *Millennium Journal of Humanities and Social Sciences*, 2023.

[38] ETS.org. Gre home, 2020.

[39] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *International Conference on Program Comprehension*, 2018.

[40] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23:452–489, 2018.

[41] Z. Fang, M. Endres, T. Zimmermann, D. Ford, W. Weimer, K. Leach, and Y. Huang. A four-year study of student contributions to oss vs. oss4sg with a lightweight intervention. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 2023.

[42] B. Floyd, T. Santander, and W. Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017.

[43] D. Ford, M. D. Storey, T. Zimmermann, C. Bird, S. Jaffe, C. S. Maddila, J. L. Butler, B. Houck, and N. Nagappan. A tale of two cities: Software developers working from home during the COVID-19 pandemic. *ACM Trans. Softw. Eng. Methodol.*, 31(2):27:1–27:37, 2022.

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

[45] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *ISSTA*, pages 177–187. ACM, 2012.

[46] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313, 2013.

[47] W. Groeneveld, H. Jacobs, J. Vennekens, and K. Aerts. Non-cognitive abilities of exceptional software engineers: A delphi study. In J. Zhang, M. Sherriff, S. Heckman, P. A. Cutter, and A. E. Monge, editors, *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, March 11-14, 2020*, pages 1096–1102, Portland, OR, USA, 2020. ACM.

[48] W. Groeneveld, L. Luyten, J. Vennekens, and K. Aerts. Exploring the role of creativity in software engineering. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Society, ICSE (SEIS) 2021, May 25-28, 2021*, pages 1–9, Madrid, Spain, 2021. IEEE.

[49] P. Guo. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In J. Nichols, R. Kumar, and M. Nebeling, editors, *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, pages 1235–1251. ACM, 2021.

[50] P. J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In T. Camp, P. T. Tymann, J. D. Dougherty, and K. Nagel, editors, *The 44th ACM Technical Symposium on Computer Science Education, SIGCSE 2013, Denver, CO, USA, March 6-9, 2013*, pages 579–584. ACM, 2013.

[51] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. DeepFix: Fixing common C language errors by deep learning. In *Conference on Artificial Intelligence*, pages 1345–1351, 2017.

[52] Y. Huang, X. Liu, R. Krueger, T. Santander, X. Hu, K. Leach, and W. Weimer. Distilling neural representations of data structure manipulation using fMRI and fNIRS. In *International Conference on Software Engineering*, pages 396–407, 2019.

[53] Y. Ikutani and H. Uwano. Brain activity measurement during program comprehension with nirs. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 2014.

[54] M. Jiménez, M. Piattini, and A. Vizcaíno. Challenges and improvements in distributed software development: A systematic review. *Advances in Software Engineering*, 2009, 2009.

[55] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[56] L. Kelion. Fbi 'could hire hackers on cannabis' to fight cybercrime. `https://www.bbc.com/news/technology-27499595`, 5 2014. Accessed: 2021-03-07.

[57] S. Keyhani, S. Steigerwald, J. Ishida, M. Vali, M. Cerdá, D. Hasin, C. Dollinger, S. R. Yoo, and B. E. Cohen. Risks and benefits of marijuana use: a national survey of us adults. *Annals of internal medicine*, 169(5):282–290, 2018.

[58] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, pages 802–811, 2013.

[59] M. Klickstein. Does pot enhance your ability to code? `https://www.itechpost.com/articles/6198/20130307/pot-enhance-ability-code.html`, 7 2013. Accessed: 2021-01-07.

[60] M. A. Kowal, A. Hazekamp, L. S. Colzato, H. van Steenbergen, N. J. van der Wee, J. Durieux, M. Manai, and B. Hommel. Cannabis and creativity: highly potent cannabis impairs divergent thinking in regular cannabis users. *Psychopharmacology*, 232(6):1123–1134, 2015.

[61] N. Kriegeskorte, M. Mur, and P. A. Bandettini. Representational similarity analysis-connecting the branches of systems neuroscience. *Frontiers in systems neuroscience*, page 4, 2008.

[62] E. Kroon, L. Kuhns, and J. Cousijn. The short-term and long-term effects of cannabis on cognition: recent advances in the field. *Current Opinion in Psychology*, 38:49–55, 2021. Cannabis.

[63] R. Krueger, Y. Huang, X. Liu, T. Santander, W. Weimer, and K. Leach. Neurological divide: An fmri study of prose and code writing. In *International Conference on Software Engineering*, 2020.

[64] E. M. LaFrance and C. Cuttler. Inspired by mary jane? mechanisms underlying enhanced creativity in cannabis users. *Consciousness and Cognition*, 56:68–76, 2017.

[65] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In *34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[66] C. M. Lee, C. Neighbors, C. S. Hendershot, and J. R. Grossbard. Development and preliminary validation of a comprehensive marijuana motives questionnaire. *Journal of studies on alcohol and drugs*, 70(2):279–287, 2009.

[67] C. Levinson. Comey: Fbi 'grappling' with hiring policy concerning marijuana. https://www.wsj.com/articles/BL-LB-48089, 5 2014. Accessed: 2021-03-07.

[68] A. Li, M. Endres, and W. Weimer. Debugging with stack overflow: Web search behavior in novice and expert programmers. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training*, pages 69–81, 2022.

[69] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, pages 101–112, 2008.

[70] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic. A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In *Agile Processes in Software Engineering and Extreme Programming: 20th International Conference, XP 2019, Montréal, QC, Canada, May 21–25, 2019, Proceedings 20*, pages 227–243. Springer International Publishing, 2019.

[71] N. Maiden, S. Robertson, and J. Robertson. Creative requirements: Invention and its role in requirements engineering. In *Proceedings of the 28th International Conference on Software Engineering*, page 1073–1074, New York, NY, USA, 2006. Association for Computing Machinery.

[72] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. Sapfix: automated end-to-end repair at scale. In H. Sharp and M. Whalen, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–278. IEEE / ACM, 2019.

[73] L. E. Margulieux. Spatial encoding strategy theory: The relationship between spatial skill and stem achievement. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 81–90, 2019.

[74] J. Markoff. *What the dormouse said: How the sixties counterculture shaped the personal computer industry*. Penguin Group, New York, NY, USA, 2005.

[75] A. McCarthy. Most popular MITx MOOC reaches 1.2 million enrollments. In *http://news.mit.edu/2018/first-mitx-mooc-reaches-enrollment-milestone-0830*, 2018.

[76] P. McLean and R. N. Horspool. A faster earley parser. In *International Conference on Compiler Construction*, pages 281–293. Springer, 1996.

[77] M. Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr, 2018.

[78] S. Moser and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9):45–51, 1996.

[79] L. Murphy, S. Fitzgerald, R. Lister, and R. McCauley. Ability to'explain in plain english'linked to proficiency in computer-based programming. In *Proceedings of the ninth annual international conference on International computing education research*, pages 111–118, 2012.

[80] E. R. Murphy-Hill, C. Jaspan, C. Sadowski, D. C. Shepherd, M. Phillips, C. Winter, A. Knight, E. K. Smith, and M. Jorde. What predicts software developers' productivity? *IEEE Trans. Software Eng.*, 47(3):582–594, 2021.

[81] T. Nakagawa, Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, and D. M. German. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *Companion proceedings of the 36th international conference on software engineering*, pages 448–451, 2014.

[82] National Academies of Sciences, Engineering, and Medicine. *The Health Effects of Cannabis and Cannabinoids: The Current State of Evidence and Recommendations for Research*. The National Academies Press, Washington, DC, 2017.

[83] K. Newman, M. Endres, W. Weimer, and B. Johnson. From organizations to individuals: Psychoactive substance use by professional programmers. In *International Conference on Software Engineering*, pages 665–677, 2023.

[84] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2228–2240, 2022.

[85] S. Ouhbi and N. Pombo. Software engineering education: Challenges and perspectives. In *2020 IEEE Global Engineering Education Conference (EDUCON)*, pages 202–209. IEEE, 2020.

[86] M. Parker, A. Solomon, B. Pritchett, D. Illingworth, L. Margulieux, and M. Guzdial. Socioeconomic status and computer science achievement: Spatial ability as a mediating variable in a novel model of understanding. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 97–105, 08 2018.

[87] M. C. Parker, M. Guzdial, and S. Engleman. Replication, validation, and use of a language independent cs1 knowledge assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, page 93–101, 2016.

[88] J. Parkinson and Q. Cutts. Investigating the relationship between spatial skills and computer science. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 106–114, 2018.

[89] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[90] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund. Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 524–536, 2021.

[91] N. Peitek, A. Bergum, M. Rekrut, J. Mucke, M. Nadig, C. Parnin, J. Siegmund, and S. Apel. Correlates of programmer efficacy and their link to experience: a combined EEG and eye-tracking study. In A. Roychoudhury, C. Cadar, and M. Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 120–131. ACM, 2022.

[92] M. Peters and C. Battista. Applications of mental rotation figures of the shepard and metzler type and description of a mental rotation stimulus library. *Brain and cognition*, 66(3):260–264, 2008.

[93] C. S. Prat, T. M. Madhyastha, M. J. Mottarella, and C.-H. Kuo. Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports*, 10(1):1–10, 2020.

[94] S. Rajasekaran and M. Nicolae. An error correcting parser for context free grammars that takes less than cubic time. In *Language and Automata Theory and Applications*, pages 533–546. Springer, 2016.

[95] J. A. Roberts, I.-H. Hann, and S. A. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52(7):984–999, 2006.

[96] P. Rodeghero and C. McMillan. An empirical study on the patterns of eye movement during summarization tasks. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[97] M. M. T. Rodrigo and R. S. J. de Baker. Coarse-grained detection of student frustration in an introductory programming course. In *International Workshop on Computing Education Research*, pages 75–80, 2009.

[98] O. Rogeberg and R. Elvik. The effects of cannabis intoxication on motor vehicle collision revisited and revised. *Addiction*, 111(8):1348–1359, 2016.

[99] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, jan 1968.

[100] G. Sakkas, M. Endres, B. Cosman, W. Weimer, and R. Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–30, 2020.

[101] G. Sakkas, M. Endres, P. J. Guo, W. Weimer, and R. Jhala. Seq2parse: neurosymbolic parse error repair. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1180–1206, 2022.

[102] P. Santiesteban, M. Endres, and W. Weimer. An analysis of sex differences in computing teaching evaluations. In *Proceedings of the Third Workshop on Gender Equality, Diversity, and Inclusion in Software Engineering*, pages 84–87, 2022.

[103] D. Shah. A product at every price: A review of MOOC stats and trends in 2017. In *Edsurge.com*, 2018.

[104] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389, 2014.

[105] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[106] S. Sorby, N. Veurink, and S. Streiner. Does spatial skills instruction improve stem outcomes? the answer is 'yes'. *Learning and Individual Differences*, 67:209–222, 2018.

[107] S. A. Sorby and B. J. Baartmans. The development and assessment of a course for enhancing the 3-d spatial visualization skills of first year engineering students. *Journal of Engineering Education*, 89(3):301–307, 2000.

[108] S. A. Sorby, E. Nevin, A. Behan, E. Mageean, and S. Sheridan. Spatial skills as predictors of success in first-year engineering. In *IEEE Frontiers in Education Conference*, pages 1–7, 2014.

[109] M.-A. Storey, T. Zimmermann, C. Bird, J. Czerwonka, B. Murphy, and E. Kalliamvakou. Towards a theory of software developer job satisfaction and perceived productivity. *IEEE Transactions on Software Engineering*, 47(10):2125–2142, 2019.

[110] U. N. P. Team. Unodc world drug report 2020: Global drug use rising; while covid-19 has far reaching impact on global drug markets, 2020.

[111] A. E. Tew and M. Guzdial. Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 97–101, 2010.

[112] R. A. Thompson. Language correction using probabilistic grammars. *IEEE Transactions on Computers*, 100(3):275–286, 1976.

[113] A. Trendowicz and J. Münch. Factors influencing software development productivity—state-of-the-art and industrial experiences. *Advances in computers*, 77:185–241, 2009.

[114] D. H. Uttal, N. G. Meadow, E. Tipton, L. L. Hand, A. R. Alden, C. Warren, and N. S. Newcombe. The malleability of spatial skills: A meta-analysis of training studies. *Psychological bulletin*, 139(2):352, 2013.

[115] P. van der Spek, N. Plat, and C. Pronk. Syntax error repair for a java-based parser generator. *ACM SIGPLAN Notices*, 40(4):47–50, 2005.

[116] T. Van Green. Americans overwhelmingly say marijuana should be legal for recreational or medical use. *Pew Research Center*, 2021.

[117] S. Wagner and M. Ruhe. A systematic review of productivity factors in software development. *arXiv preprint arXiv:1801.06475*, 2018.

[118] M. Walton. Programming and cannabis — 5 things to know. `https://simpleprogrammer.com/programming-and-cannabis/`, 4 2019. Accessed: 2021-03-07.

[119] C. Wilcox and A. Lionelle. Quantifying the benefits of prior programming experience in an introductory computer science course. In *Proceedings of the 49th acm technical symposium on computer science education*, pages 80–85, 2018.

[120] L. Wu, F. Li, Y. Wu, and T. Zheng. Ggf: A graph-based method for programming language syntax error correction. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 139–148, 2020.

[121] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In E. Bodden, W. Schäfer,

A. van Deursen, and A. Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 740–751. ACM, 2017.