Predictability of identifier naming with Copilot: A case study for mixed-initiative programming tools

Michael Jing Long Lee
Computer Laboratory
University of Cambridge
mjll2@cam.ac.uk

Advait Sarkar
Microsoft Research
advait@microsoft.com

Alan F. Blackwell
Computer Laboratory
University of Cambridge
Alan.Blackwell@cl.cam.ac.uk

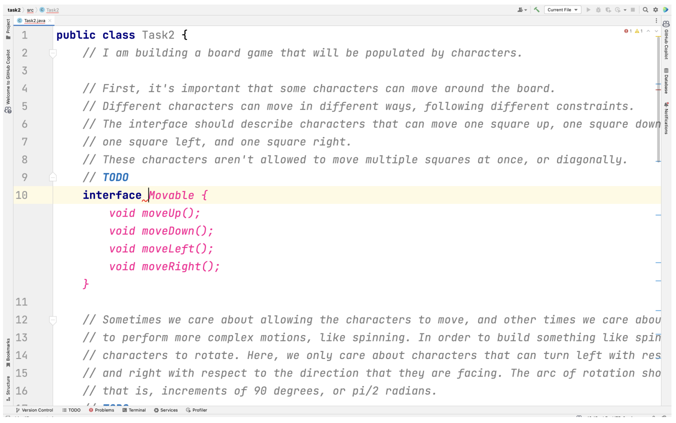
Abstract

Studies show that predictive text entry systems make writing faster, but written content more predictable. We consider if these trade-offs extend to code synthesis tools such as GitHub Copilot. While Copilot can make developers produce code faster, it may also affect how they choose identifiers for methods and classes. This may have non-trivial effects on the activity of programming, because identifier names are a primary semantic signal in code, and play important roles in authoring, debugging, and developer communication. In a controlled, within-subjects experiment (n=12), we compared identifiers chosen in the presence and absence of Copilot suggestions. We find that identifiers chosen in the presence of Copilot suggestions were significantly more predictable (have lower mean entropy), even when suggestions were only visible and could not be automatically accepted. These results imply that mixed-initiative systems can take an active role in shaping programmer intentions and potentially impact their sense of agency. We consider whether an increased convergence towards predictable names is an asset or a liability for the practice of programming, and suggest design opportunities for surfacing surprising identifiers and conceptual refactoring tools.

1. Introduction

Code synthesis tools based on generative Large Language Models (LLMs), such as GitHub Copilot (hence **Copilot**), have been widely adopted by developers and firms (Dohmke, 2023). In February 2023, Copilot was estimated to produce 46% of code across all programming languages, a percentage that had doubled over the previous year (Zhao, 2023). Unlike traditional code-completion tools, Copilot has the ability to suggest multiple lines of code at once, and to recommend potential identifiers (Ziegler et al., 2022). It has been found to increase productivity – both actual (Mozannar et al., 2023), and perceived (Peng et al., 2023; Ziegler et al., 2022).

We consider these programmer aids from the perspective of *mixed-initiative interaction*, in which either the user or the tool might take the next action. This means that the system must make judgments on how



```java
public class Task2 {
    // I am building a board game that will be populated by characters.

    // First, it's important that some characters can move around the board.
    // Different characters can move in different ways, following different constraints.
    // The interface should describe characters that can move one square up, one square down
    // one square left, and one square right.
    // These characters aren't allowed to move multiple squares at once, or diagonally.
    // TODO
    interface Movable {
        void moveUp();
        void moveDown();
        void moveLeft();
        void moveRight();
    }

    // Sometimes we care about allowing the characters to move, and other times we care abou
    // to perform more complex motions, like spinning. In order to build something like spin
    // characters to rotate. Here, we only care about characters that can turn left with res
    // and right with respect to the direction that they are facing. The arc of rotation sho
    // that is, increments of 90 degrees, or pi/2 radians.
```

Figure 1 – An example study task as seen by participants, with a suggestion by Copilot highlighted in pink. Participants were more likely to accept identifier suggestions (in this case, Movable), than to generate original names (e.g., TakesSingleStep or Move).

well it understands the user's goals, and on when it might be appropriate to interrupt the user with an offer of assistance (Horvitz, 1999).

In the case of programming tools such as Copilot, the utility function for mixed initiative interaction is less easily calculated, because the only explicit "goal" of a working programmer is the system specification, itself often ambiguous or incomplete. More tractably, the programmer's goal from moment to moment is simply to refine their understanding of the problem domain and of the required execution behaviour (Naur, 1985), expressing that developing model with conceptual clarity and economy, for example by well-chosen identifier names.

In this paper, we specifically consider the choice of identifier names as a valuable case study through which to understand the nature of interaction with Copilot from a mixed initiative perspective. Choosing good names for identifiers is a key skill of the working programmer (McConnell, 1993). For example, the name of an abstract type often reflects basic concepts in an application domain, a function name should succinctly describe the operation that will be performed, and a field in a database schema might express an important feature of a customer relationship. For programmers developing reusable frameworks, libraries and APIs, the name of each element is critical to the usability of the whole (Furnas et al., 1987).

As a result, defining, reviewing and updating identifier names is an essential conceptual element of programming work, in which the programmer both refines and communicates their understanding of the software engineering problem in a way that will be understandable by other programmers (Schankin et al., 2018).

Mixed-initiative interaction when choosing identifier names can be considered as a trade-off in attention investment (Blackwell, 2002). Mixed-initiative interaction and the attention investment model are both fundamentally about the cost-benefit tradeoff of automation (Williams et al., 2020). In this case, Copilot might suggest a conventional identifier name at relatively low attentional cost, but the programmer could alternatively invest attention in making the name more informative and specific to a distinctive context they are working in (Blackwell, 2022).

We relate this to prior work, showing that in certain cases, intelligent user interfaces that streamline *processes* to be more efficient can also make *content* more generically predictable (Arnold et al., 2020), and therefore less informative (Shannon, 1948). A common situation in machine learning-based code assistants is that the system may propose a conventional name based on its training corpus. This works well in highly standardised programming tasks such as student coding exercises, and also in very routine or conventional programming work where a single correct name might be highly predictable. The challenge that we address here comes in situations where a programming task is not standardised, perhaps in a new domain or involving an original approach. In those cases where the best name cannot be straightforwardly predicted from prior code, reuse of conventional identifiers could easily misrepresent the programmer's intention and be subtly incorrect. Our investigation therefore focuses on this tradeoff between originality and predictability, recognising that each has its place in good quality code.

We adapt the experimental paradigm developed by Arnold et al. (2020) for study of predictive text, applying their approach in the context of source code identifiers. The authors of that study take care to note that their conclusions did not necessarily extend to tasks involving conceptual exposition. In contrast, the creation of identifiers, which describe the properties of abstract objects, is exactly the task of conceptual exposition. Our controlled, within-subjects experiment (n=12) compared identifier naming with and without Copilot support, including a condition where suggestions were only visible, but not available as automated actions.

The main contribution of this work is to demonstrate a statistically reliable effect, that the visible presence of Copilot suggestions results in more predictable identifiers, which may sometimes be desirable, but not in more novel domains or coding tasks. We consider the consequences of this phenomenon in relation to attention investment for mixed-initiative programming tools, and suggest design strategies that might mitigate the problems that can result where predictable or conventional code is not a primary quality objective.

2. Related Work

2.1. Predictability of AI-assisted work and critical integration

The theory of *critical integration* (Sarkar, 2023b) is a general account of the nature of generative AI-assisted knowledge workflows. According to this, as the work of material production (e.g., the physical typing of text or code, or creation of images) is increasingly delegated to AI, the role of the user is to critically evaluate and integrate AI output into their broader workflow. However, the workflow itself and the user's objectives can be affected through interaction with AI output.

For example, models of interaction with predictive text systems (Bhat et al., 2023) have identified specific cognitive processes (Hayes, 2012) that are influenced by suggestions. Notably, the writer's respect for the system affects the degree to which suggestions are accepted. Additionally, suggestions shape Working Memory State. Therefore, they impact not only syntactic choices, but sentence structure and semantic content. Suggestions have even been found to influence authors' topic choices and opinions (Jakesch et al., 2023; Poddar et al., 2023).

Arnold et al. (2020)'s work is theoretically grounded in Rational Speech Act (RSA), a goal-oriented model of communication. Under RSA, speakers choose phrases by balancing utility and cost (Goodman & Frank, 2016). If words are chosen whilst writing (MacArthur et al., 2016), reducing the cost of a predictable word can prompt users to choose less informative phrases.

Buschek et al. (2021) and Singh et al. (2023) show how such findings may be operationalised, by designing predictive text interfaces that leverage cognitive impacts to aid ideation. Proposals generally involve encouraging users to *critically* integrate suggestions. They include surfacing multiple suggestions at once, and forcing explicit integration of suggestions rather than automatic acceptance.

2.2. Attention Investment

Good identifier names involve an attention investment (Blackwell, 2003): by *investing* immediate attention, programmers may choose a distinctive name that accurately summarises hundreds of lines of original code. In doing so, there is a *pay-off*: future attentional cost savings, since they or others may efficiently surmise the nature of the abstract object by its name. However, there is also a *risk* that no pay-off accrues, which varies depending on the nature of the task.

The attention investment problem is especially acute in settings where identifier names are hard to conceptualise but have the potential to be very informative (cf: Section 4.1). As identified by Blackwell (2022), such settings include

1. Naming concepts that are frequently *reused*, potentially in different settings,
2. Naming concepts that are *related*, for example, in API design, where related methods chain to form a language, and
3. Naming *refactored* concepts, where updated requirements or semantics are reflected in subtle changes to names (Blackwell, 2023).

2.3. Studies of GitHub Copilot

A systematic review of research on GitHub Copilot identified developer productivity, code quality, code security, and education as primary themes (Ani et al., 2023). Evaluation of Copilot as a mixed-initiative system tends to define utility in terms of productivity impact (Mozannar et al., 2023; Peng et al., 2023). While the effect of Copilot on identifier choice has not been considered, results show that Copilot increases productivity. However, as Buschek et al. (2021) found, ideation and efficiency are often in tension, so greater volume of code production may be associated with more conventional or homogeneous names.

Multiple evaluations of Copilot have found that while it is useful in some situations, it requires the programmer to still exercise algorithmic thinking, program comprehension, debugging and communication skills, and can prove a liability for non-expert programmers (Dakhel et al., 2023; Fajkovic & Rundberg, 2023; Imai, 2022; Zhang et al., 2023b). These have led researchers to caution against indiscriminate use of AI assistance in programming education settings (Puryear & Sprint, 2022; Wermelinger, 2023). Moreover, while the complexity and readability of Copilot-generated code is comparable to that written by humans, eye-tracking data suggests that programmers pay less visual attention to AI-generated code (Al Madi, 2022), corresponding to studies of agency in mixed-initiative interaction where users are less critical of automated suggestions when they perceive the machine as having greater agency (Yu et al., 2021).

Benchmark tests show that performance of GitHub Copilot, OpenAI ChatGPT, and Amazon CodeWhisperer can approach human level, but varies depending on the target language (Nguyen & Nadi, 2022; Yetistiren et al., 2022; Yetiştiren et al., 2024). Inappropriate sensitivity to the prompting language is also a challenge; in one study Copilot generated different code results for semantically equivalent natural language prompts in approximately 46% of the test cases (Mastropaolo et al., 2023). Moreover, while Copilot can be prompted in multiple natural languages, it is not equally performant, with one study finding that performance with Chinese language prompts was significantly worse than with English (Koyanagi et al., 2024).

Studies on developers' subjective experience (Kalliamvakou, 2023; Sarkar et al., 2022; Vaithilingam et al., 2022; Vasconcelos et al., 2023; Zhang et al., 2023a; Zhou et al., 2023) and mental models (Mozannar et al., 2022) have additionally found that Copilot reduces perceived mental effort and that users often accept suggestions without verification, which they defer to some future point. Such deferrals, as well as the introduction of suboptimal solutions or unaddressed issues which can interfere with future software development, can contribute to technical debt (OBrien et al., 2024). Tools such as Copilot can be used to facilitate the authoring of code when programmer intent is clear, but also to aid exploration and discovery (Barke et al., 2023; Sarkar et al., 2022). While Copilot can improve efficiency, it can come at the cost of code comprehension and autonomy or control (Bird et al., 2022). An analysis of a corpus of software developers' tweets about GitHub Copilot found that programmers' negative emotions can become more positive when the capabilities of the AI tools are linked to their identity work (Eshraghian et al., 2023). When considered in the framework of attention investment, these both hint at less attention being invested into identifier names.

3. Research Questions

We aim to understand how developers are influenced by the identifiers suggested by Copilot. If developers tend to *accept* Copilot's suggestions, this may result in more predictable identifier names (the assumption being that Copilot produces more predictable names, an assumption which we discuss in Section 6). We also consider whether making it more effortful to accept suggestions, by disabling keyboard shortcuts for easy acceptance, can affect the influence of Copilot on identifier names (and thereby programmer agency). Our research questions are:

RQ1: To what extent are identifiers more predictably named in the presence of Copilot suggestions?

RQ2: To what extent do results differ if keyboard shortcuts for accepting suggestions are disabled?

4. Study Design

To evaluate the effect of Copilot suggestions on identifier choice, we conducted a within-subjects experiment in which participants completed short Java programming tasks (Section 4.1) under different levels of access to Copilot suggestions (Section 4.2). The 12 participants were computer science undergraduates at our institution, recruited via convenience sampling. All participants had prior knowledge of Java interfaces and experience in practical Java programming through undergraduate-level coursework.

Expression	Definition	Interpretation
C	A set of common concepts named by participants	NA
$\text{names}(c,t)$	The names given to concept c under treatment t	NA
$H_{\text{names}}(c,t)$	The entropy of $\text{names}(c,t)$	The *unpredictability* of the names given to a **specific** c under t
$\{H_{\text{names}}(c,t) \mid c \in C\}$	The set of all $H_{\text{names}}(c,t)$ in a given treatment t	Assuming $H_{\text{names}}(c,t)$ is independent of c, this estimates the *distribution* of $H_{\text{names}}(t)$
$\langle H_{\text{names}}(t) \rangle$	The mean of $\{H_{\text{names}}(c,t) \mid c \in C\}$	The predictability of the names given to an **arbitrary** c under t

Table 1 – Collated Definitions

4.1. Tasks

Using IntelliJ IDEA, participants defined Java interfaces based on natural language prompts. Three tasks were developed, each with the aim of reflecting a context where distinctive original names are useful, but hard to conceptualise.

Task 1 involved defining interfaces that form a pipeline for working with data. Participants had to consider the *relationships* between interfaces, and methods that could be *reused* in a variety of contexts. For example, a method for `checking` data could be called by a process writing to, or reading from, a database. The checks could differ in the two cases.

Task 2 involved defining interfaces for a game, where characters could move around a grid, and rotate in-place. Careful naming was required to capture the *relationships* between interfaces, for example, methods to move `right` and to rotate `right` might clarify if the motion is *relative* or *absolute*.

Task 3 involved participants developing a structure for managing custom user settings. Participants were asked to imagine that this was originally a command line tool, that was being replaced by a GUI. This *refactoring* task encouraged participants to consider how the changing context updates requirements, and how these updates may be reflected in changes to existing names.

The prompts were designed to avoid priming participants to pick certain identifier names adopting the method described in Liu and Sarkar, et al. (Liu et al., 2023). Tasks were described in verbose and indirect ways, encouraging participants to make new identifier choices rather than reuse vocabulary from the task descriptions.

In the original study of predictive text by Arnold et al. (2020), the experimental task involved writing image captions. The predictive text system was allowed to consider the image prompt as part of the context when generating suggestions. By analogy to that study, we included the prompt stimulus text in comment blocks so that it was visible to Copilot. Copilot may also consider content in other open files. To ensure all participants saw the same initial suggestions, the set of open files was controlled.

A full listing of our experimental tasks and prompts is given in Appendix A.

4.2. Treatments

We manipulate the visibility of suggestions and the mechanism for accepting suggestions, resulting in three conditions:

1. ON: Copilot is enabled, with keyboard shortcuts for accepting suggestions.

2. VIEW: Copilot is enabled, but keyboard shortcuts were disabled. Users could view the suggestion, but could only incorporate it in their code by manually typing it out.

3. OFF: Copilot is disabled, and no suggestions were shown.

Many IDEs, including IntelliJ IDEA, have native (non-AI based) code completion tools that are widely used in practice. These tools offer autocomplete so that programmers can repeatedly reference identifiers already present in the codebase. Because the autocomplete functionality acts as a confounding factor in this setting and interfere with programmer's attention towards Copilot suggestions, code completion was disabled for all three treatments to preserve internal validity. This comes at a slight cost to external validity, but as Intellij IDEA's native code completion tool does not suggest potential new identifiers, and our tasks did not require participants to reference the same identifier multiple times, its absence is unlikely to have been detrimental.

4.3. Protocol

The study was carried out in-person. All 12 participants declared that they were familiar with programming in Java, and read and signed a statement of informed consent. Participation in the study was voluntary and participants were not directly compensated. Our study protocol was approved by our institution's ethics committee.

Participants were first asked to read a description of the study, which explained that they would be asked to define interfaces under three different treatments, and that the experiment was a study of Copilot. We did not explicitly draw attention to identifiers, but asked participants to consider the readability and maintainability of their code.

Before attempting any of the tasks, participants were first given a tutorial, where they familiarised themselves with defining interfaces and working with Copilot. Participants then completed each of the three tasks in turn. The assignments of tasks to conditions was counterbalanced, so that each task was completed by 4 participants each in the ON, OFF, and VIEW conditions respectively. The study sessions lasted between 45-60 minutes.

ds : An interface for reading and writing data			
	$t \in T$		
	$t =$ ON	$t =$ VIEW	$t =$ OFF
$\mathrm{names}(\mathsf{ds}, t)$	DataSource, DataSource, DataSource, DataSource	DataSource, DataSource, DataSource, DataSource	Datum, GetAndSet, Manipulator, QueryData
$H(\cdot)$	0	0	2

Table 2 – Example Computation of $H_{names}(\mathsf{ds}, t)$. $H_{names}(\mathsf{ds}, t)$ is the entropy of the distribution of names (each column) given to ds *under treatment $t \in T$.*

4.4. Measures

Since all participants were given the same three task descriptions, all participants were creating names relating to the described set of concepts C.[1] As a running example, consider one such concept described in Task 1 — ds: an interface that reads and writes to some source of data. As shown in Table. 2, we consider $\mathrm{names}(\mathsf{ds}, t)$: the bag of names given by participants to ds under treatment $t \in \{$ON, VIEW, OFF$\}$.

To measure predictability, we employ Shannon Entropy, an information theoretic model for quantifying the average amount of information communicated by a source (Shannon, 1948). By measuring average surprisal, entropy quantifies unpredictability.

We ask: "What did programmer X name concept ds under treatment t?". $H_{\mathrm{names}}(\mathsf{ds}, t)$ quantifies the *uncertainty* of the answer. Mathematically, it is the entropy of the empirical distribution of the bag. If

[1]Some concepts were not named by all participants. In particular, participants disagreed on how to encode inputs to functions, with some choosing not to specify them at all. These concepts were excluded.

the bag has only one unique element, the name can be predicted with certainty; the entropy is 0. In general, an entropy of n can be interpreted as the uncertainty associated with predicting the name from one of 2^n equiprobable candidates. This increases as predictability decreases.

To generalise from a single concept ds to the effect of a treatment t on an *arbitrary* concept c, we make the simplifying assumption that $H_{\text{names}}(c,t)$ is independent of c (not generally the case, but reflecting our experimental tasks). Hence, each $H_{\text{names}}(c,t)$ is an observation of the same random variable, $H_{\text{names}}(t)$, and $\{H_{\text{names}}(c,t) \mid c \in C\}$ is a sample from the underlying distribution. Let $\langle H_{\text{names}}(t) \rangle$ denote the sample mean. This is the *expected* unpredictability of an identifier under t, regardless of the concept it names. These definitions are collated in Table 1.

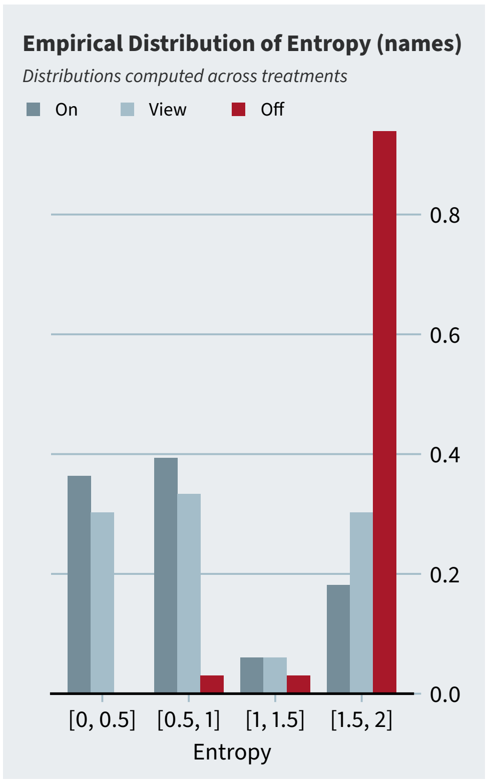
This analysis was repeated at the *word* level: $H_{\text{words}}(\text{ds},t)$. This was to investigate whether Copilot encourages multiword identifiers that reshuffle words drawn from a smaller vocabulary. Finally, we noted cases where participants changed their first choice, effectively renaming the identifier, since revisiting a previous decision represents additional investment of attention by the namer.

We analysed the effect of each treatment on predictability. Two levels of granularity were considered: a fine-grained comparison of entropy *distributions* was reinforced by a coarse-grained comparison of *means*. 95% confidence intervals were estimated by bootstrap re-sampling with replacement (1000 iterations).

5. Results

5.1. Predictability

Fig. 2 plots histograms of the sample $\{H(c,t) \mid c \in C\}$ for each treatment t, as an estimate of the underlying distribution of $H(t)$.



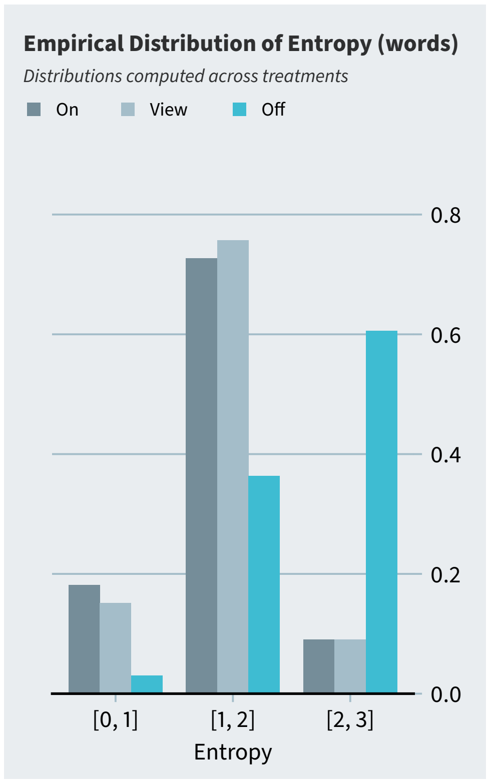


Figure 2 – Treating each $H(c,t)$ as an observation of $H(t)$, these histograms estimate the underlying distribution of $H(t)$ for each treatment t. Left: $H_{\text{names}}(t)$ and Right: $H_{\text{words}}(t)$ [2]

[2]Bin sizes were chosen for interpretability.

Fig. 3 illustrates the sample mean $\langle H(t) \rangle$ for each treatment $t \in T$. [3] Fig. 4 illustrates, for pairs of treatments $(t_1, t_2) \in T \times T$, the pairwise difference $\langle H(t_1) \rangle - \langle H(t_2) \rangle$.

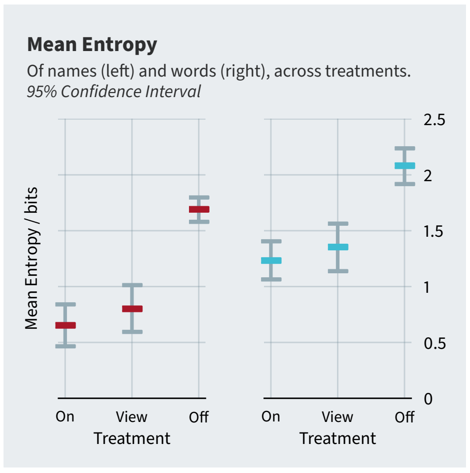


Figure 3 – Mean entropy $\langle H(t) \rangle$ for each treatment t. This is the mean of $H(c,t)$ (Table 2) over all concepts c under the same treatment t. Left: $\langle H_{names}(t) \rangle$. Right: $\langle H_{words}(t) \rangle$.

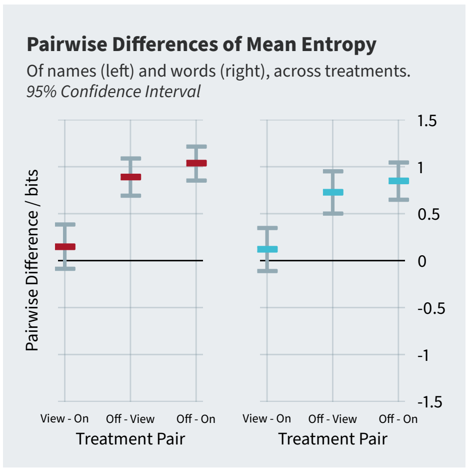


Figure 4 – Difference in mean entropy $\langle H(t_1) \rangle - \langle H(t_2) \rangle$ for pairs of treatments (t_1, t_2). Left: name level. Right: word level.

At the name level, the mean entropy was 1.04 bits higher when Copilot was OFF compared to the ON treatment (CI: $[0.81, 1.19]$), and 0.90 bits higher than the VIEW treatment (CI: $[0.69, 1.09]$).

Fisher's Exact Test confirmed this to be a statistically significant difference. Under the null hypothesis

$$\mathscr{H}_0 : \langle H_{\text{names}}(\text{OFF}) \rangle = \langle H_{\text{names}}(\text{VIEW}) \rangle$$

$\mathscr{H}_0$ was rejected at the $p = 0.01$ level ($p = 0.001$).

At the name level, the mean entropy was 0.15 bits higher under the VIEW treatment as compared to the ON treatment (CI: $[-0.13, 0.36]$). This is not statistically significant at the 0.01 level ($p = 0.18$).

Analysis at the word level revealed similar results. The mean entropy of words was 0.85 bits higher when Copilot was OFF compared to the ON treatment, (CI: [0.65, 1.05]), and 0.73 bits higher than the VIEW treatment. (CI: [0.51, 0.96]). The mean entropy under the VIEW treatment was 0.12 bits higher than under the OFF treatment. Symmetric to the name level results, this difference between ON and OFF was significant at the $p = 0.01$ level ($p = 0.001$) but the difference between VIEW and OFF was not ($p = 0.26$).

Table 3 presents the same data in terms of predictability rather than entropy. Given an arbitrary concept c, Fig. 3 tabulates the probability that c is more unpredictably named under the OFF treatment than the ON and VIEW treatments. More precisely, this is the probability that for some fixed concept c, $H_{\text{names}}(c, \text{OFF}) > H_{\text{names}}(c,t)$, $t \in \{\text{ON}, \text{VIEW}\}$. If given a random concept c, it is likely that c is more unpredictably named under the OFF treatment than under the ON ($p = 0.91$, CI: $[0.82, 1.00]$) or VIEW treatments ($p = 0.85$, CI: $[0.73, 0.97]$).

5.1.1. Probability of Renaming

When Copilot was OFF, participants renamed their "initial" identifier – defined as the first identifier they typed – with probability 0.26. Under the VIEW and OFF treatments, where the "initial identifier" is defined as Copilot's suggestion, the probability of renaming dropped to 0.20 and 0.11 respectively (Table 4). While the difference in probability between the OFF and ON treatments is significant at the $\alpha = 0.05$ level ($p = 0.04$), the difference between OFF and VIEW treatments is not ($p = 0.06$).

[3] Mean entropy is higher for **words** than **names** because each name is counted as multiple words. Hence, an outlier name can be counted as three or four outlier words.

t	$P(H(c,\text{OFF}) > H(c,t))$	95% CI
ON	0.909	$[0.818, 1.000]$
VIEW	0.848	$[0.727, 0.970]$

Table 3 – Probability that a randomly drawn concept is more predictably named under the ON *and* VIEW *treatment than under the* OFF *treatment, with 95% CI. (Unpredictability quantified by entropy of the empirical set of names).*

t	$P(\text{renamed}\|t)$	95% CI
ON	0.106	$[0.061, 0.160]$
VIEW	0.197	$[0.129, 0.267]$
OFF	0.258	$[0.182, 0.333]$

Table 4 – Probability that a participant re-named the "initial" name for a concept under treatment t. If $t \in \{\text{ON}, \text{VIEW}\}$, the "initial" name is Copilot's suggestion. If $t = \text{OFF}$, the "initial" name is the first name typed by participants.

5.1.2. Interfaces vs. Methods

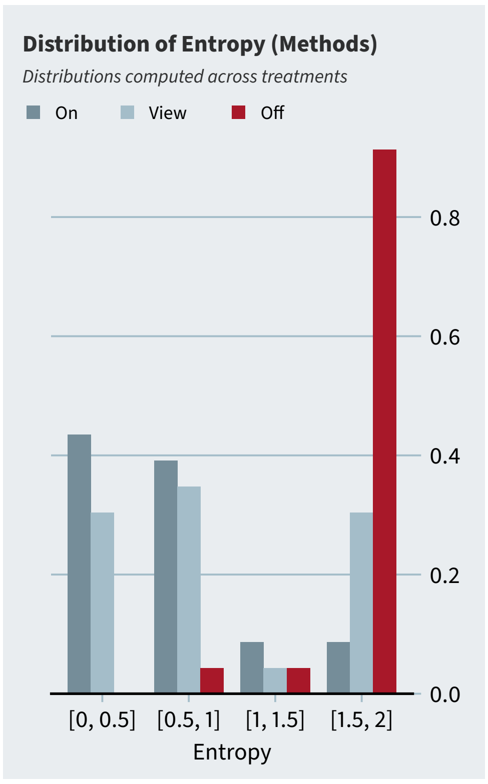
Our experimental tasks required participants to name two distinct types of concepts: interfaces and methods. Fig. 5 compares the distribution of entropy for interfaces, $H_{\text{names}}(i,t)$, with the distribution of entropy for methods, $H_{\text{names}}(m,t)$. Under the ON treatment, the mean entropy of interface names is 0.39 bits higher than for method names. However, this is not statistically significant ($p = 0.06$). Under the VIEW treatment, the difference between the entropy of interfaces and methods is -0.01 bits, and under the OFF treatment, the difference between the entropy of interfaces and methods is 0.08 bits.

6. Discussion

We find that, regardless of the mechanism for accepting suggestions (**RQ2**), names are significantly more predictable in the presence of Copilot suggestions (**RQ1**). Under the ON and VIEW treatments, for more than 67% of concepts, less than 1 bit of information was needed to determine the chosen name. This means that concepts were *more* predictably named than if all participants were given the same two names, and asked to pick one at random. Under the OFF treatment, for more than 90% of concepts, $H_{\text{names}}(c, \text{OFF})$ was greater than 1.5 bits. As only four participants named each concept under each treatment, the maximum possible entropy is 2 bits (4 unique names), i.e., the empirically observed diversity in the OFF condition approaches the theoretical limit.

Our quantitative and qualitative data support the interpretation that participants experienced an attention investment trade-off in identifier naming with Copilot. Three participants indicated that they felt they could have improved on Copilot's suggestions, but "*it just wasn't worth the effort*" (P2, P6, P10). This suggests that participants felt that the marginal attentional cost of improving on Copilot's suggestion was higher than that of improving on one's own candidate name. This is corroborated by the experimental data, which showed that participants were more than twice as likely to re-name their initially chosen identifiers when Copilot was OFF than the suggested identifier when Copilot was ON.

However, this is not a complete explanation, as the difference in the probability of re-naming under the OFF and VIEW treatments was not significant. This could be attributed to two factors. First, the process of typing out Copilot's suggestion reduced the marginal cost of thinking up a better name. Second, we underestimated the frequency of renaming under the OFF treatment, by assuming that the first identifier written down by participants was the first candidate name considered. Hence, if participants considered several names before typing out an identifier, which they did not later edit, this was *not* counted as a re-naming. In contrast, under the VIEW and OFF treatments, the frequency of re-naming could be measured much more accurately.



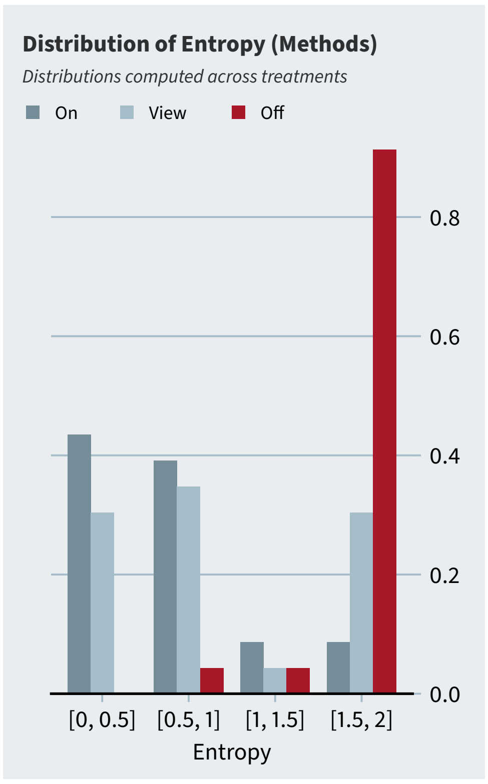


Figure 5 – Left: Distributions of entropy that only consider interfaces and ignore methods: $H_{names}(i,t)$. *Right: Distributions of entropy that only consider methods and ignore interfaces:* $H_{names}(m,t)$.

6.1. Mixed-initiative systems, agency, and mechanised convergence

These results have implications for our understanding of how contemporary mixed-initiative programming tools can introduce much broader concerns than the traditional narrow focus on task completion. In particular, our findings suggest that there may be implications for mixed-initiative interaction on agency as well as the convergence (homogeneity) of output.

For example, one participant who stated that they "*care a lot*" (P2) about naming noted that programming with Copilot ON was harder than with it OFF, as they felt like they were "*fighting to break free*" of the names suggested by Copilot. Yu et al. (2021) showed that mixed-initiative systems can cause users to feel a loss of agency, which may increase cognitive load. Darvishi et al. (2024) found that AI assistance impacts the agency of students, causing them to rely on rather than learn from AI. While Kalliamvakou (2023) posits that Copilot reduces cognitive load by automating mundane tasks, our results suggest that should developers decide to invest attention into a task, such as choosing a good name, Copilot may decrease feelings of agency and thus increase cognitive load. The perception of agency is an important aspect of the user experience in interacting with intelligent text assistants (Yu et al., 2023), and convergence to Copilot naming might reduce the overall agency and ownership perceived by the programmer. Sarkar (2023a) observes that in generative AI-assisted end-user programming, the traditional attention investment trade-off (between the costs of automation, the time saved, and the risks of failing to build a useful automation) may well be subsumed by considerations of agency and trust in automation.

The second challenge posed by our findings is to the idea that mixed-initiative systems neutrally progress users towards achieving their goals. When the goal is broad and admits a variety of solutions (as in the case of identifier naming), the system may actually influence the goal rather than just infer it. This may or may not be inappropriate – in cases where the programmer should be using a standardised solution or algorithm, but has not recognised this, substitution of a more conventional, predictable, identifier could improve their solution. However, in aspects of software development that relate to contextual and domain understanding, standardised solutions may be worse.

Consider the mixed-initiative nature of traditional code completion tools (Mărășoiu et al., 2015), and paradigms such as programming by demonstration (Cypher & Halbert, 1993) or programming by example (Lieberman, 2001), and compare their properties to Copilot. Previous work has largely focused on the technical challenge of inferring the user's goals, over which the user is assumed to have complete autonomy. In contrast, here we observe that the mixed-initiative system is taking an active role in goal-shaping. And the particular form of goal-shaping we have observed in our study corresponds to the phenomenon of *mechanised convergence* (Sarkar, 2023b).

Mechanised convergence is a general principle positing that automation has a standardisation effect, reducing the frequency of outliers. For example, a study of consultants at Boston Consulting Group found that ideas generated with AI assistance had a *"marked reduction in ... variability ... compared to those not using AI. ... it might lead to more homogenized outputs"* (Dell'Acqua et al., 2023). Similarly, Anderson et al. (2024) found that *"different users tended to produce less semantically distinct ideas with ChatGPT"* and further, that this could impact agency: *"ChatGPT users ... felt less responsible for the ideas they generated"*.

In the context of creating identifier names, the principle of mechanised convergence suggests that as names become more predictable, this reduces the frequency of very bad names, but also the frequency of very good ones. One researcher informally analysed the identifiers authored during the study for informativeness. With Copilot OFF, there were more extremely informative, and extremely uninformative identifiers. For example, consider the task where participants were asked to name a character that can move around a grid, one square at a time. With Copilot ON, most participants chose the name `Movable` – this is moderately informative as it states what can be done with the character but contains no information about the one-square constraint. With Copilot OFF, the quality of names ranged from `Move` (very bad) to `TakesSingleStep` (very good). `Move` is uninformative, as the vocative case of the verb "to move" is more appropriate for a function that causes the character to move, and the noun form indicating a specific instance of a motion (i.e., in the sense of "a dance move") is more appropriate for an object that records a move instance. Both senses of `Move` fail to describe the character's ability (unlike the adjective `Movable`), and also fail to capture the one-square constraint. On the other hand, `TakesSingleStep` is extremely informative, uses an appropriate grammatical form, and captures the one-square constraint. A full listing of identifiers written by our participants by task and condition is given in Appendix B.

Is mechanised convergence, *per se*, an asset or a liability for the practice of programming? Even if Copilot's suggested identifiers cannot match the quality or informativeness of those written by the best programmers, they only need to be better than those written by *most* programmers for the aggregate benefits of naming-by-Copilot to outweigh the negatives. However, programmers do not experience the practice of programming in aggregate (Bergström & Blackwell, 2016), and individual programmers almost certainly vary in their naming skill at different times and in different contexts. Moreover, we must also consider not simply the quality of the final identifier, but also the cognitive challenges and benefits of inventing it. The process of naming a concept itself might induce changes or insights. For example, one craft practice of programming holds that if a function is hard to name, this is probably an indication that one is doing too much or too little in that function (Blackwell et al., 2008).

When Copilot suggestions were enabled, suggestions for method names were more readily accepted by participants than suggestions for interface names. Participants occasionally thought of names for interfaces while reading the problem description, *before* seeing Copilot's suggestion, but this was rare for methods. Even without Copilot suggestions, the predictability of names may vary between concepts. Concepts for which strong conventions exist – for example, getters, setters, and common algorithms like QuickSort – might be named more predictably than bespoke methods or interfaces. However, in all treatments the mean entropy for interfaces does not differ significantly from the mean entropy for methods. Hence, there is insufficient evidence to suggest that interfaces are more, or less, predictably named than methods.

6.2. Implications for design and developer practice

While Github Copilot may boost developer productivity, it also results in significantly more predictable identifiers. This may be because Copilot suggestions increase the attentional costs of improving on a suggested identifier.

These findings offer suggestions for developer workflows that increasingly require "critical integration" (Sarkar, 2023b) of Copilot-generated code.

First, consider settings where good names are costly, but important. For example, when establishing a new set of naming conventions for a codebase. Given that *appropriate* names suggested by Copilot increase the marginal cost of investing attention, the converse might also hold: *inappropriate* suggestions may decrease this cost, and encourage programmers to think more carefully about names, a similar strategy to Wilson et al. (2003)'s *Surprise-Explain-Reward* model, in which the user's attention is drawn toward features of the code that they didn't expect.

Second, consider settings where good names are not as critical. For example, when an established convention already exists, and predictable names are informative within the context. In these cases, Copilot may help developers follow existing conventions in predictable ways. In turn, this may help create a setting where unpredictable names draw attention more effectively. When the predictability of a set of names increases, an outlier is more surprising. Hence, when most names are predictable, deliberate breaks from convention can more effectively emphasise subtle differences and direct developers' attention.

We can also draw on the observations from this empirical study to suggest several design opportunities for mixed-initiative features that could result in improved quality of identifier names.

First, it is important to note that in some cases, the predictable names suggested by Copilot might sometimes be better names than more idiosyncratic alternatives created by the programmer. This may be because the programmer's suggestion reflects a misunderstanding of the problem, or perhaps a lack of knowledge of standard approaches. In these cases, it could be beneficial to the programmer to invest more attention, thinking again about the reason for their name choice. Wilson et al. (2003)'s *Surprise-Explain-Reward* design pattern can help here, alerting the programmer to the unconventional name they have chosen, and giving them the opportunity to investigate why this is the case.

A second design opportunity could be to optimise investment of attention with better understanding of contextual factors that are relevant to naming, such as distinguishing between a) throw-away programming "sketches" where the code will be discarded immediately after execution; b) systems intended to have a long maintained lifetime that will involve intermittent attention from many different programmers; or c) API libraries and frameworks where thousands of programmers will eventually need to understand the implications of the identifiers chosen. In cases where the choice of identifier names has especially costly implications, a programming assistance tool would be able to take this into account by collecting information about the eventual audience and context of use, encoding that contextual information as additional prompts to the LLM during code generation.

A third design opportunity is to consider a new kind of software development / maintenance tool that might be described as "conceptual refactoring", which makes no changes to the function or semantics of the source code, but simply modifies identifier names. During incremental and iterative software development, it is not unusual for programmers to improve their understanding of the system such that they see opportunities to improve on the identifier names that were initially chosen. A conceptual refactoring tool, by focusing only on identifier names, could improve the overall clarity and coherence of that name space. Technical strategies that might achieve this through the use of LLMs could include use of summarising approaches to extract and clarify the variety of identifiers that have been used in a large code base, then reconsidering individual identifiers in terms of their role within that overall structure. Such a tool could be implemented as direct interaction with a symbol table or data dictionary, or by using chat dialog prompts such as "Please include in your response a list of identifiers you've used, with the reasons for your choices" (Lewis, 2024).

6.3. Limitations and Future Work

Sample Size. While the results we have observed are statistically significant, it is also possible (given the upper bound on the entropy), that the effect size has been underestimated. Supplemental analyses that increase the sample size might find an even larger effect.

Assumption of Independence. The simplifying assumption that the predictability of an identifier depends only on the treatment, not the concept, may be loosened. While we took preliminary steps in this direction - broadly dividing concepts between interfaces and methods - further work could consider finer granularity in these subdivisions. As suggested, Copilot might have a smaller effect on the predictability of getter and setter names than other methods.

External Validity. The decision to disable IntelliJ IDEA's code completion tool could be revisited. The presence of IntelliJ IDEA's code completion tool could have been manipulated as an independent factor, resulting in three more treatments. While this was not feasible due to resource constraints, it represents a natural extension to the study.

Mechanised Convergence. A surface-level survey of the names finds results consistent with the phenomenon of "mechanised convergence". However, a more thorough analysis requires considering not only changes to the *predictability* of names, but *quality* of names, including cases where the best quality name *would* be a very predictable one (for example when implementing a standard algorithm such as quicksort). Analysis might involve multiple raters ranking names by quality in context, with consistency achieved by employing set-wise comparison (Sarkar et al., 2016).

Sample Homogeneity. We only studied CS undergraduates at one University. Undergraduates are, in general, less than experienced software engineers. They may find it more difficult to choose good names, and be more susceptible to authoritative suggestions. Programmers in industry may be trained to respect certain company-specific conventions, or constraints, in naming. Future work may consider whether the effects generalize to samples of professional programmers.

Language Effects. This study considered the effect of Copilot on identifier names *in Java* specifically. Different programming languages are used by different types of programmer, and for different purposes, which may bring different implications for attention investment. As a result, distinct languages often have distinct conventions for identifiers. Future work might consider if and how the effect on identifier choice varies between languages. For languages with larger training corpora, e.g., popular languages such as Python and JavaScript, identifiers that follow conventions may be assigned higher probabilities by the language model, and so the effect size is unlikely to vary.

7. Conclusion

This study explored how AI code generation tools like GitHub Copilot influence the conceptual task of choosing identifiers during programming. Selecting descriptive names for classes, methods, and variables is a crucial activity that shapes code readability and communicates intent. Yet developers may face tensions between investing sufficient attention for informative naming versus prioritizing efficiency.

We conducted a controlled experiment where 12 participants defined Java interfaces both with and without the presence of Copilot's identifier suggestions. Across three coding tasks carefully designed to require subjective naming decisions, identifiers chosen under Copilot's influence were found to have significantly lower entropy – that is, they were more predictable and less informative. Strikingly, this tendency towards predictable names occurred even when Copilot merely displayed suggestions without allowing auto-completion.

We find that generative AI problematizes the traditional task-oriented narrative of mixed-initative systems. Mixed-initiative systems can have an impact on programmer agency as well as their goals. While predictable names promote consistency, overly deferring to AI suggestions could deprioritise investing the human attention required to craft identifiers that are specifically tailored to the nuances of the current context and requirements.

To mitigate risks of AI prematurely narrowing programmers' perspectives, we propose AI tools that surface surprising or unconventional alternatives, counterbalancing predictable suggestions. Incorporating conceptual refactoring aids could also encourage revising identifiers as the programmer's understanding evolves.

As AI's role in programming extends beyond just accelerating tasks, this work underscores the need to thoughtfully steer AI-assisted workflows. Simply optimizing for productivity could inadvertently discourage essential cognitive activities that underpin coding quality. Balancing AI assistance with preserving key human skills like intentional naming will be crucial.

Acknowledgments
Thanks to our participants for their valuable time.

References

Al Madi, N. (2022). How readable is model-generated code? examining readability and visual inspection of github copilot. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–5.

Anderson, B. R., Shah, J. H., & Kreminski, M. (2024). Homogenization effects of large language models on human creative ideation.

Ani, Z. C., Hamid, Z. A., & Zhamri, N. N. (2023). The recent trends of research on github copilot: A systematic review. *International Conference on Computing and Informatics*, 355–366.

Arnold, K. C., Chauncey, K., & Gajos, K. Z. (2020). Predictive text encourages predictable writing. *IUI '20: Proceedings of the 25th International Conference on Intelligent User Interfaces*. `https://doi.org/10.1145/3377325.3377523`

Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, *7*(OOPSLA1), 85–111.

Bergström, I., & Blackwell, A. F. (2016). The practices of programming. *2016 ieee symposium on visual languages and human-centric computing (vl/hcc)*, 190–198.

Bhat, A., Agashe, S., Oberoi, P., Mohile, N., Jangir, R., & Joshi, A. (2023). Interacting with next-phrase suggestions: How suggestion systems aid and influence the cognitive processes of writing. *IUI '23: Proceedings of the 28th International Conference on Intelligent User Interfaces*. `https://doi.org/10.1145/3581641.3584060`

Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., & Gazit, I. (2022). Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, *20*(6), 35–57.

Blackwell, A. F. (2002). First steps in programming: A rationale for attention investment models. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2–10.

Blackwell, A. F. (2003). First steps in programming: A rationale for attention investment models. *IEEE*. `https://doi.org/10.1109/hcc.2002.1046334`

Blackwell, A. F. (2022, September). Chapter 10: The craft of coding [https://moralcodes.pubpub.org/pub/chapter-9]. In *Moral Codes*. MIT Press.

Blackwell, A. F. (2023, June). Chapter 11: How can stochastic parrots help us code? [https://moralcodes.pubpub.org/pub/1osz744d]. In *Moral Codes*. MIT Press.

Blackwell, A. F., Church, L., & Green, T. R. (2008). The abstract is an enemy: Alternative perspectives to computational thinking. *PPIG*, 5.

Buschek, D., Zürn, M., & Eiband, M. (2021). The impact of multiple parallel phrase suggestions on email input and composition behaviour of native and non-native english writers. *CHI '21: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. `https://doi.org/10.1145/3411764.3445372`

Cypher, A., & Halbert, D. C. (1993). *Watch what i do: Programming by demonstration*. MIT press.

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. J. (2023). Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, *203*, 111734.

Darvishi, A., Khosravi, H., Sadiq, S., Gašević, D., & Siemens, G. (2024). Impact of ai assistance on student agency. *Computers Education*, *210*, 104967. `https://doi.org/https://doi.org/10.1016/j.compedu.2023.104967`

Dell'Acqua, F., McFowland III, E., Mollick, E. R., Lifshitz-Assaf, H., Kellogg, K., Rajendran, S., Krayer, L., Candelon, F., & Lakhani, K. R. (2023, September). *Navigating the jagged technological frontier: Field experimental evidence of the effects of ai on knowledge worker productivity and quality* (Working Paper No. 24-013). Harvard Business School Technology Operations Mgt. Unit. `https://doi.org/10.2139/ssrn.4573321`

Dohmke, T. (2023, June). The economic impact of the ai-powered developer lifecycle and lessons from github copilot - the github blog. `https://github.blog/2023-06-27-the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/`

Eshraghian, F., Hafezieh, N., Farivar, F., & De Cesare, S. (2023). Dynamics of emotions towards ai-powered technologies: A study of github copilot. *Academy of Management (AOM) Annual Meeting 2023*.

Fajkovic, E., & Rundberg, E. (2023). The impact of ai-generated code on web development: A comparative study of chatgpt and github copilot.

Furnas, G. W., Landauer, T. K., Gomez, L. M., & Dumais, S. T. (1987). The vocabulary problem in human-system communication. *Communications of the ACM*, *30*(11), 964–971.

Goodman, N. D., & Frank, M. C. (2016). Pragmatic language interpretation as probabilistic inference. *Trends in Cognitive Sciences*, *20*(11), 818–829. https://doi.org/10.1016/j.tics.2016.08.005

Hayes, J. R. (2012). Modeling and remodeling writing. *Written Communication*, *29*(3), 369–388. https://doi.org/10.1177/0741088312451260

Horvitz, E. (1999). Principles of mixed-initiative user interfaces. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 159–166.

Imai, S. (2022). Is github copilot a substitute for human pair-programming? an empirical study. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 319–321.

Jakesch, M., Bhat, A., Buschek, D., Zalmanson, L., &, (2023). Co-writing with opinionated language models affects users' views. *CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3544548.3581196

Kalliamvakou, E. (2023, September). Research: Quantifying github copilot's impact on developer productivity and happiness - the github blog. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

Koyanagi, K., Wang, D., Noguchi, K., Kondo, M., Serebrenik, A., Kamei, Y., & Ubayashi, N. (2024). Exploring the effect of multiple natural languages on code suggestion using github copilot. *arXiv preprint arXiv:2402.01438*.

Lewis, C. (2024).

Lieberman, H. (2001). *Your wish is my command: Programming by example*. Morgan Kaufmann.

Liu, M. X., Sarkar, A., Negreanu, C., Zorn, B., Williams, J., Toronto, N., & Gordon, A. D. (2023). "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. *CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3544548.3580817

MacArthur, C. A., Graham, S., & Fitzgerald, J. (2016, October). *Handbook of writing research, second edition*. Guilford Publications.

Mărășoiu, M., Church, L., & Blackwell, A. F. (2015). An empirical investigation of code completion usage by professional software developers. *Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group*.

Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., & Bavota, G. (2023). On the robustness of code generation techniques: An empirical study on github copilot. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2149–2160.

McConnell, S. (1993, May). *Code complete: A practical handbook of software construction*. http://ci.nii.ac.jp/ncid/BA26593422

Mozannar, H., Bansal, G., Fourney, A., & Horvitz, E. (2022). Reading between the lines: Modeling user behavior and costs in ai-assisted programming. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2210.14306

Mozannar, H., Bansal, G., Fourney, A., & Horvitz, E. (2023). When to show a suggestion? integrating human feedback in ai-assisted programming. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2306.04930

Naur, P. (1985). Programming as theory building. *Microprocessing and microprogramming*, *15*(5), 253–261.

Nguyen, N., & Nadi, S. (2022). An empirical evaluation of github copilot's code suggestions. *Proceedings of the 19th International Conference on Mining Software Repositories*, 1–5.

OBrien, D., Biswas, S., Imtiaz, S., Abdalkareem, R., Shihab, E., & Rajan, H. (2024). Are prompt engineering and todo comments friends or foes? an evaluation on github copilot. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 1003–1003.

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of ai on developer productivity: Evidence from github copilot. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2302.06590

Poddar, R., Sinha, R., & Jakesch, M. (2023). Ai writing assistants influence topic choice in self-presentation. *CHI EA '23: Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3544549.3585893

Puryear, B., & Sprint, G. (2022). Github copilot in the classroom: Learning to code with ai assistance. *Journal of Computing Sciences in Colleges*, *38*(1), 37–47.

Sarkar, A. (2023a). Will Code Remain a Relevant User Interface for End-User Programming with Generative AI Models? *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 153–167. https://doi.org/10.1145/3622758.3622882

Sarkar, A. (2023b). Exploring Perspectives on the Impact of Artificial Intelligence on the Creativity of Knowledge Work: Beyond Mechanised Plagiarism and Stochastic Parrots. *CHIWORK '23: Proceedings of the 2nd Annual Meeting of the Symposium on Human-Computer Interaction for Work*. https://doi.org/10.1145/3596671.3597650

Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Srinivasa Ragavan, S., & Zorn, B. (2022). What is it like to program with artificial intelligence? *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022)*.

Sarkar, A., Morrison, C., Dorn, J. F., Bedi, R., Steinheimer, S., Boisvert, J., Burggraaff, J., D'Souza, M., Kontschieder, P., Rota Bulò, S., Walsh, L., Kamm, C. P., Zaykov, Y., Sellen, A., & Lindley, S. (2016). Setwise Comparison: Consistent, Scalable, Continuum Labels for Computer Vision. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 261–271. `https://doi.org/10.1145/2858036.2858199`

Schankin, A., Berger, A., Holt, D. V., Hofmeister, J. C., Riedel, T., & Beigl, M. (2018). Descriptive compound identifier names improve source code comprehension. *2018 ACM/IEEE 26th International Conference on Program Comprehension*. `https://doi.org/10.1145/3196321.3196332`

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, *27*(3), 379–423. `https://doi.org/10.1002/j.1538-7305.1948.tb01338.x`

Singh, N., Bernal, G., Savchenko, D., & Glassman, E. L. (2023). Where to hide a stolen elephant: Leaps in creative writing with multimodal machine intelligence. *ACM Transactions on Computer-Human Interaction*, *30*(5), 1–57. `https://doi.org/10.1145/3511599`

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. `https://doi.org/10.1145/3491101.3519665`

Vasconcelos, M. H., Bansal, G., Fourney, A., Liao, Q. V., & Vaughan, J. (2023). Generation probabilities are not enough: Exploring the effectiveness of uncertainty highlighting in ai-powered code completions. *arXiv (Cornell University)*. `https://doi.org/10.48550/arxiv.2302.07248`

Wermelinger, M. (2023). Using github copilot to solve simple programming problems, 172–178. `https://doi.org/10.1145/3545945.3569830`

Williams, J., Negreanu, C., Gordon, A. D., & Sarkar, A. (2020). Understanding and Inferring Units in Spreadsheets. *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 1–9. `https://doi.org/10.1109/VL/HCC50065.2020.9127254`

Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C. R., Durham, M. D., & Rothermel, G. (2003). Harnessing curiosity to increase correctness in end-user programming. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. `https://doi.org/10.1145/642611.642665`

Yetistiren, B., Ozsoy, I., & Tuzun, E. (2022). Assessing the quality of github copilot's code generation. *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*, 62–71.

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2024). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. arxiv preprint arxiv: 230410778. 2023. *arXiv preprint arXiv:2304.10778*.

Yu, C. G., Blackwell, A. F., & Cross, I. (2021). Perception of rhythmic agency for conversational labeling. *Human-Computer Interaction*, *38*(1), 25–48. `https://doi.org/10.1080/07370024.2021.1877541`

Yu, C. G., Blackwell, A. F., & Cross, I. (2023). Perception of rhythmic agency for conversational labeling. *Human–Computer Interaction*, *38*(1), 25–48.

Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023a). Demystifying practices, challenges and expected features of using github copilot. *arXiv preprint arXiv:2309.05687*.

Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023b). Practices and challenges of using github copilot: An empirical study. *arXiv preprint arXiv:2303.08733*.

Zhao, S. (2023, February). Github copilot now has a better ai model and new capabilities - the github blog. `https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/`

Zhou, X., Liang, P., Zhang, B., Li, Z., Ahmad, A., Shahin, M., & Waseem, M. (2023). On the concerns of developers when using github copilot. *arXiv preprint arXiv:2311.01020*.

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A. S., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2022). Productivity assessment of neural code completion. *MAPS 2022: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. `https://doi.org/10.1145/3520312.3534864`

Appendices

A. Experimental Prompts and Tasks

A.1. Prompts

This experiment is interested in how programmer style is influenced by the use of CoPilot.

It will comprise three tasks. Each task will require you to define one or more interfaces or abstract classes.

Unless explicitly told otherwise, you will not need to implement the interfaces, nor implement any of the methods in the abstract classes. In essence, the focus of task is on the interface/class signatures, not on the logic.

Each task will be in its own `.java` file, with a comment explaining the nature of the task. Points where you are expected to write code are explicitly flagged with `TODO`.

Each task will have a different CoPilot configuration. There are three configurations:

1. One where CoPilot is turned on,
2. One where you can see CoPilot's suggestions, but you can't automatically accept them (you have to type them out), and
3. one where CoPilot is off.

The researcher will adjust the settings for each of the tasks. Please do not modify them.

When you are done with the task, please indicate to the researcher that you are happy with your submission, at which point the researcher will change the CoPilot settings, and allow you to move to the next task.

If you have any questions, please indicate them to the researcher at this point.

A.2. Tasks

A.2.1. Warmup

```java
interface TwoDimPoint {
    int getX();
    int getY();
    void setX(int v);
    void setY(int v);
}

// TODO: Extend this interface to obtain a ThreeDimPoint interface.
```

A.2.2. Task 1

```java
public class Task1 {
    // A data pipeline is a series of steps for working with data. This typically
    // involves reading, extracting, transforming, manipulating, validating,
    // checking, visualising, storing, plotting, and many other steps.

    // You are tasked with designing three interfaces that, if implemented,
    // form a very basic pipeline for working with data.

    // First, an interface that is able to ask some source, or knowledge object,
    // or database for some datum or data. Further, it should be able to take some
    // datum or data and ask the source/object/database/knowledge base to store it.
    // When this interface is given a datum or data to store, it should
    // automatically assume that the datum or data is valid. Further, you can
    // assume  that any singular piece of datum or data will be encoded as a string
    .
    // TODO

    // We have the data from the source, or to be written into the source.
    // What's next? Well, we want an interface that makes sure we're not doing
    // anything silly. The interface should have some signal that can call on when
    // the checks have failed. This ought to be an exception. Now, we assume all
    // users are internal, so throwing this checked exception shouldn't crash the
    // program, but instead be caught and handled. This means we want checked
    // exceptions to be thrown. Your job is to define some checked exception that
    // is thrown when the second interface catches some mangled or nasty data.
    // TODO

    // Now we are ready to define the second interface.
    // This interface should have two roles.
    // Its first role is to take a datum or data that the first interface has
    // returned, and check it for faults.
    // These flaws could be mistakes, or corruption, or garbled data. While not
    // important, potential sources of  corruption could be faults in the hardware,
    // problems in transmission over the wire, etcetera. Its second role is that
    // it should have a method that checks if the data that the user gives to the
    // first interface is valid. So this treats data going in the other direction.
    // The causes of the potential mistakes/corruption are different in this case,
    // as the user could have made a mistake, by encoding some flawed data, or by
    // calling a buggy encoding method.
    // TODO

    // Now we are ready to define the third interface.
    // The final task is to give clients something that they can actually use.
    // So far we've only been working with data encoded as strings.
    // Clients don't actually want the strings, but they want something that's
    // easier to play around with. This involves molding or reshaping the data
    // into something they can actually use. We don't want to rely on the clients
    // building proper decoders, so it makes sense to define a couple of defaults
    // that they can call on to manipulate the data into the form that they
    // actually plan on using. We've done some studies, and we've narrowed down
    // the three most commonly used formats to be:
    //      json,
    //      byte64, and
    //      a Map object
    // Hence, the interface should expose three methods, one for each format.
    // TODO
}
```

A.2.3. Task 2

```
public class Task2 {
    // I am building a board game that will be populated by characters.

    // First, it's important that some characters can move around the board.
    // Different characters can move in different ways, following different
    // constraints. The interface should describe characters that can move
    // one square up, one square down, one square left, and one square right.
    // These characters aren't allowed to move multiple squares at once, or
    // diagonally.
    // TODO

    // Sometimes we care about allowing the characters to move, and other
    // times we care about allowing the characters to perform more complex
    // motions, like spinning. In order to build something like spinning, we
    // need to allow characters to rotate. Here, we only care about characters
    // that can turn left with respect to the direction that they are facing,
    // and right with respect to the direction that they are facing. The arc
    // of rotation should be a quarter-circle, that is, increments of 90 degrees,
    // or pi/2 radians.
    // TODO

    // Third, some characters will have an inventory.
    // An inventory is a collection of items that a character can carry around
    // with them. We care about characters that are able to take a single item
    // from their surroundings, and put it into their inventory. After they do
    // so, they will be able to carry these items around with them.
    // Characters should also be able to retrieve stuff from the inventory, to
    // use it or discard it in some manner. Characters should operate on singular
    // items, that is, there won't be a method to put many things into the
    // inventory, or take many things out of it, in just a single go.
    // TODO

    // Fourth, some characters will be able to throw things.
    // When a character throws something, they will throw it in the direction
    // that they are facing, rather than in any arbitrary direction. This means
    // that they can only throw things in the direction they are facing. In
    // addition, characters need to throw something. They can’t just throw nothing.
    // Further, characters that implement this interface should throw things
    // exactly 5 squares. To recap: characters can throw things 5 squares in the
    // direction that they are facing.
    // TODO

    // As a test, build a Character interface that extends each of the previous
    // interfaces
    // TODO

    // Build a dodge method that moves a character. Assume that the character is
    // facing the up direction.You want to move it in a zig-zag motion in the up
    // direction. The character should turn to face the direction of motion and
    // moving in some interesting pattern. The signature of the method should be
    // void dodge(Character c)
    // TODO

    // Build an attack method: get a stone from the inventory, throw it, walk 5
    // steps, and get the stone back. The signature of the method should be
    // void attack(Character c)
    // TODO

    // Finally, build an interface for characters that can move 5 squares up,
    // down, left, and right. They shouldn’t be able to move in increments of
    // less than 5.
    // TODO
}
```

A.2.4. Task 3

```
public class Task3 {
    /*
     We have a user-facing command line system.
     Users can personalise the system, for example, changing the time zone, language, and font size.

     There are also some global settings, or defaults. These global settings / defaults kick in
     when the user has not specified any personal settings. For example, when the user is a new user.
     In some sense these settings are pre-installed or pre-defined by the company, though once the
     software has shipped, users (be they people or companies) can change these global or default settings
     to their liking.

     Currently, the mechanism for changing settings, both global and local, is by setting feature flags.
     A feature flag is a single bit, indicating if the feature is on, or off.
     The code will query these feature flags to determine methods to execute, or items to display.
     This means that the system will operate differently depending on if a feature flag is on, or off.
     Specifically, this is done via a command line tool, called, flg. There are 5 ways to use flg

     flg 0 is a getter, that returns the user’s custom settings. For example, if the user settings are
     "1011", then flg 0 will return "1011" for that user. Different users will get
    different results.

     flg 0[sequence] acts as a setter for the user's private, personal, custom
    settings.
     The sequence is used to determine what settings ought to be set.
     For example, flg 011001 turns feature flag 0 on, feature flag 1 on, feature
    flag 2 off, feature flag 3 off,
     and feature flag 4 on.

     flg 1 is a getter, and it gets the global or universal settings. For example,
    if the global settings are 0000, then
     flg 1 will return 0000.

     flg 1[sequence] changes the global settings (for everyone). The sequence is
    used to determine what the new
     global settings ought to be. For example, flg 10111 will turn feature flag 1,
    feature flag 2, and feature flag 3 on,
     and it will turn feature flag 0 off. It does not delete any custom settings.
    That is, if the user already has
     feature flag 0 on, they will not notice a change.

     flg 2 deletes all custom settings, effectively resets all custom settings to
    the global default. So for example,
     if there are 3 users, users A, B, and C, and they each have custom settings,
    and the global setting is 0000, then
     after we call flg 2, all 3 users will have 0000 as their settings.

     The company has decided to move to a system with a GUI.
     You have been asked to refactor flg into an abstract class (no implementation
    required, all methods should be stubs).
     This class will not be exposed to the user via a command line interface.
     This means there is no need for backwards compatibility, and you only need to
    preserve the functionality, not
     the exact syntax, or the exact mechanisms that supply the functionality.
     Indeed, you have been advised to define one method for each possible different
     way to use flg.
     Your abstract class should store the global settings as a static list of
    integers.
     You abstract class should also store the user's custom settings as a static
    dictionary from user ID (string) to a list of integers.
     */
    // TODO

}
```

B. Participant Responses

Tasks are named Tn::Im::Mk, as in Task n, Java Interface m, Java Method k.

	On	View	Off
T1::I1	DataSource	DataSource	ManipulateData
	DataSource	DataStore	GetAndSettable
	DataSource	DataSource	SourceQuery
	DataSource	DataSource	Datum
T1::I1::M1	query	read	getData
	read	getData	get
	read	read	retrieve
	read	read	getDatum
T1::I1::M2	store	store	storeData
	write	storeData	set
	write	store	store
	write	write	storeDatum

Continued on next page

	On	View	Off
T1::I2::E1	DataValidationException	DataIntegrityException	CheckedException
	DataException	DataHandlerException	SillyData
	DataException	DataException	SourceException
	DataException	DataException	MangledDataException
T1::I2::E2	DataValidationException	DataConsistencyException	CheckedException
	DataException	DataHandlerException	SillyData
	DataException	DataException	SourceException
	DataException	DataException	MangledDataException
T1::I3	Validator	DataChecker	CheckData
	DataChecker	DataHandler	DataChecker
	DataChecker	DataValidater	SourceVerifier
	DataChecker	DataChecker	DataChecker
T1::I3::M1	validate	checkRead	checkDataOutput
	checkRead	checkDataReturn	checkRetrievedData
	checkRead	sourceValidate	checkForFaults
	checkRead	checkRead	checkReturnedData
T1::I3::M2	validate	checkWrite	checkDataInput
	checkWrite	checkDataInput	checkInputData
	checkWrite	userValidate	checkForFaults
	checkWrite	checkWrite	checkInputData
T1::I4	Decoder	DataTransformer	convertData
	DataTransformer	DataDecoder	Decoder
	DataTransformer	DataDecoder	Decoder
	DataTransformer	DataTransformer	DataManipulator
T1::I4::M1	decodeJson	toJson	jsonData
	json	asJson	toJson
	json	json	toJson
	json	json	dataToJson
T1::I4::M2	decodeBase64	toBase64	byte64Data
	byte64	asByte64	toBase64
	byte64	byte64	toByte64
	byte64	byte64	dataToByte64
T1::I4::M3	decodeMap	toMap	mapData
	map	asMap	toMap
	map	map	toMap
	map	map	dataToMap
T2::I1	Movable	TakesSingleStep	Movable
	Movable	CharacterTranslations	Movable
	Movable	CharacterMove	MovableCharacter
	Movable	Move	MovingCharacter
T2::I1::M1	moveUp	movesOneSquareUp	moveUp
	moveUp	moveUp	moveUp
	moveUp	moveUp	moveUp
	moveUp	up	moveUp
T2::I1::M2	moveDown	movesOneSquareDown	moveDown
	moveDown	moveDown	moveDown
	moveDown	moveDown	moveDown
	moveDown	down	moveDown
T2::I1::M3	moveLeft	movesOneSquareLeft	moveLeft
	moveLeft	moveLeft	moveLeft
	moveLeft	moveLeft	moveLeft
	moveLeft	left	moveLeft
T2::I1::M4	moveRight	movesOneSquareRight	moveRight
	moveRight	moveRight	moveRight
	moveRight	moveRight	moveRight
	moveRight	right	moveRight
T2::I2	Rotatable	AbleToRotate	Rotatable
	Rotatable	CharacterRotations	Rotatable
	Rotatable	CharacterRotate	RotatableCharacter
	Rotatable	Rotate	SpinningCharacter
T2::I2::M1	rotateClockwise	turnsLeft90Degrees	rotateLeft

Continued on next page

	On	View	Off
	rotateLeft	rotateLeft	rotateLeft
	rotateLeft	rotateCW	rotateLeft
	rotateLeft	rotateLeft	rotateLeft
T2::I2::M2	rotateAntiClockwise	turnsRight90Degrees	rotateRight
	rotateRight	rotateRight	rotateRight
	rotateRight	rotateACW	rotateRight
	rotateRight	rotateRight	rotateRight
T2::I3	Inventory	HasInventory	Inventory
	Inventory	CharacterInventory	Inventory
	Inventory	CharacterInventory	CharacterWithInventory
	Inventory	Inventory	InventoryCharacter
T2::I3::M1	storeItem	storeItem	takeItem
	takeItem	store	takeItem
	takeItem	store	takeItem
	storeItem	put	takeItem
T2::I3::M2	retrieveItem	retrieveItem	retrieveItem
	retrieveItem	retrieve	retrieveItem
	retrieveItem	retrieve	retrieveItem
	retrieveItem	retrieve	retrieveItem
T2::I4	Throwable	AbleToThrow	Thrower
	Thrower	CharacterThrow	Thrower
	Thrower	CharacterThrow	CharacterThatThrows
	Thrower	Throw	ThrowingCharacter
T2::I4::M1	throwItem	throwForwardFiveSquares	throwItem
	throwItem	throwFive	throwItem
	throwItem	throwFive	throwItemFiveSquares
	throwItem	throwItem	throwItem
T3::I1	FeatureFlags	Flg	Flg
	FeatureFlag	Flg	Settings
	Flg	Flg	Flg
	Flg	Flg	Flg
T3::I1::A1	defaultFlags	globalSettings	globalSettings
	global	globalSettings	globalSettings
	globalFlags	globalSettings	globalSettings
	globalSettings	globalSettings	globalSettings
T3::I1::A2	userFlags	customSettings	customSettings
	custom	userSettings	customSettings
	userFlags	customSettings	userSettings
	custom	customSettings	customSettings
T3::I1::M1	setDefault	getGlobalSettings	getGlobalSettings
	getGlobal	getGlobalSettings	getGlobalSettings
	flg1	getCustomSettings	getUserSettings
	flg0Get	getCustomSettings	getGlobalSettings
T3::I1::M2	getDefault	setGlobalSettings	setGlobalSettings
	setGlobal	setGlobalSettings	setGlobalSettings
	flg1	setCustomSettings	setUserSettings
	flg0Set	setCustomSettings	setGlobalSettings
T3::I1::M3	setUser	getCustomSettings	getCustomSettings
	getCustom	getUserSettings	getCustomSettings
	flg0	getGlobalSettings	getGlobalSettings
	flg1Get	getGlobalSettings	getCustomSettings
T3::I1::M4	getUser	setCustomSettings	setCustomSettings
	setCustom	setUserSettings	setCustomSettings
	flg0	setGlobalSettings	setGlobalSettings
	flg1Set	setGlobalSettings	setCustomSettings
T3::I1::M5	clear	reset	reset
	deleteAllCustom	reset	resetToGlobalSettings
	flg2	reset	resetUserSettings
	flg2Del	reset	deleteCustomSettings

Table 5 – Participant Responses