Co-Designing Web Interfaces for Code Comparison

Justin Middleton
North Carolina State University
Raleigh, North Carolina, USA
Email: jamiddl2@ncsu.edu

Neha Patil
North Carolina State University
Raleigh, North Carolina, USA
Email: np771999@gmail.com

Kathryn T. Stolee
North Carolina State University
Raleigh, North Carolina, USA
Email: ktstolee@ncsu.edu

2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | 979-8-3503-6613-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/VL/HCC60511.2024.00029

***Abstract*—Developers use the internet to find, learn about, and reuse code. During these processes, developers explore alternative programs whose syntactic differences may be subtle yet behavioral differences significant, and vice versa. Unfortunately, accurate comprehension is time-consuming and error-prone, to say nothing of code comparison. Given these circumstances, we run a collaborative design activity to explore how web interfaces can support better code comparison for search and reuse. We recruited 11 developers from academia and industry to discuss potential designs for three online contexts: searching, recommending, and learning. For each context, we collaboratively sketched interfaces that may support developers' present goals without the technical limitations of current approaches. We report the patterns of features and arrangements that developers want from current and future interfaces, distinguishing the statically discoverable information from the dynamically produced.**

I. Introduction

The Internet has transformed software development by overwhelming the developer with opportunity. Software need not be written from scratch with every new project; instead, a developer can consult search engines, seek peer recommendations, and learn new techniques with only a few clicks from their home page. As a result, developers become keen reusers of code and opportunistic students of algorithms. Observational and telemetric studies confirm that developers spend significant time on a typical workday foraging in its reserves, seeking fixes and reference templates over many iterations [1], [2], [3].

However, the search is not often for *the* answer as much as it is *an* answer. Many problems have multiple solutions, as anyone who has studied the efficiency trade-offs of sorting algorithms can attest. To further that example, even if one were to narrow the search to quicksorts, any two developers may have different stylistic and formatting preferences informed by their unique experiences. In other words, the search space for code is rich with syntactic differences that disguise semantic similarities, or vice versa. These patterns are apparent from the richness of similar if not outright duplicated across the internet [4], [5]. The landscape of potential code is nuanced even more by the introduction of generative AI tools, capable of remixing familiar solutions into seemingly plausible candidates for new contexts without guarantees of correctness [6].

Reuse-driven development tests the developers' ability in these two ways. First, there are more options than a developer could hope to apprehend. Second, developers must distinguish meaningful differences from the superficial along a spectrum of ambiguous similarities, which is cognitively difficult [7].

Thankfully, developers need not rely on their comprehension ability alone, as the designs of development tools and web interfaces facilitate (or frustrate) users' goals. For example, tools that analyze hyperlinks and user behavior may be able to predict what link a user will click next and emphasize that link accordingly [8]. The nature of the goals may change from website to website, but investigating what the interfaces do or do not *yet* do, we may discover interface designs that let developers compare options more effectively for their goals—and with better comparison, make better decisions in reuse.

To explore the design space of goal-directed interfaces for code comparison, we performed a collaborative interface design activity with 11 developers. We walked through their thoughts and obstacles in two of three scenarios—learning, recommending, and searching—to sketch potential designs from a hypothetical ideal or technologically-unlimited interface. This co-design methodology results in 22 sketches of interfaces that embody what developers prefer when dealing with multiple similar code alternatives. From our collaborative sketches, we derive the themes of the *features* of the website and the *arrangements* with which they contextualize each other. Patterns emerge from these themes, such as their relative priorities in determining quality code or limits on how many alternatives developers prefer to have visible at any time. Specifically, we make the following contributions:

- Categorization of website interface elements by what code-relevant information they contain and how they help developers compare code alternatives (Table III).
- Eleven barriers for participants when performing comparative comprehension on the web (Table IV).
- Recommendations and illustrations for future comparison tools to help developers' search workflows (Section V-C).

To contextualize these results, we also provide background on other efforts within the code comprehension and comparison space. Lastly, we comment on the limitations of our approach, as well as the future work, such as the continued iterative prototyping of the ideas generated alongside developers.

II. Related Work

Understanding the impact of the web interface on comparison needs background in three areas: *program comprehension*, *web-powered software development*, and *code compararison tools*.

A. Program & Comparative Comprehension

Program comprehension is the cognitive process where developers learn the structure and behavior of software. Storey surveys the significant theories of program comprehension, such as bottom-up comprehension, which begins with the bare text, and top-down comprehension, which begins with developers' prior knowledge [9]. As von Mayrhauser and Vans synthesize it, the reality of comprehension lies between these two poles [10], so developers use multiple behaviors to support cognition. Ko and colleague's model, for example, has a three-phase process of *searching* for information, *relating* new information to old, and *collecting* resources to maintain mental models [11].

When writing code, developers often reuse approaches they encountered previously. Détienne explores reuse in terms of the *schema*, or cognitive construct of an algorithm, and the *analog*, or the existing program which ostensibly relates to the same problem [12]. Likewise, Ragavan and colleagues demonstrate that developers can use the similarity of code variants to navigate codebases during reuse activities [13], [8]. Middleton and colleagues formalize this cognition as *comparative comprehension* [7]. Their work sheds light on code comparison tasks and highlights the importance of understanding the underlying factors that influence developers' accuracy in comparing code alternatives. Our research here applies comparative comprehension to specific web contexts.

B. Software Development Through the Internet

While the integrated development environment (IDE) is often at the center of programming, the development process altogether extends beyond it, as Xia and colleagues observe [3]. Sadowski and colleagues characterize developers' use of search engines for code retrieval [2]: on an average weekday, developers performed 5.3 brief but iterative search sessions, often for finding examples or sample code. These patterns were later corroborated by Bai and colleagues [14]. Furthermore, Kuttal and colleagues explore programmers' navigation of online repositories, tracing their use of information signals via interface features for searching among alternatives [15]. Our focus resembles theirs, albeit with a method that replaces observations of existing interfaces with co-design of speculative ones, and a heightened focus on explicit comparison.

Developers also frequently turn to Q&A platforms like Stack Overflow, where they can solicit recommendations and reuse code snippets in the responses [16]. In Wu and colleagues' study [17], a significant portion of the sourced code (78.2%) requires modification to suit developers' project requirements, ranging from minor adjustments to complete rewrites. This echoes Holmes and Walkers' findings on *pragmatic reuse* plans [18]. Nevertheless, in Wu's study, barriers to code reuse on Stack Overflow include the need for extensive code modification, incomprehensible or low-quality code, and challenges in integrating the sourced code into developers' projects. While the aforementioned study explores developers' experiences with the content of interfaces, our focus expands to the layout of features that could enhance developers' efficiency.

C. Interfaces for Code Comparison

Code comparison has explicit application in activities like code review, so some tools exist to support it. At the base level are text-diffing tools [19]; more sophisticated are tree-differencing algorithms, such as in Falleri and colleagues' work [20]. However, given that code review has many difficulties beyond finding changed code [21], [22], numerous additional tools can clarify the relationships between similar versions of code. For example, Huang and colleagues' CLDIFF tool labels and links edited code segments between versions to assist change comprehension [23], and Collector-Sahab augments the code comparison interface with dynamic information from runtime [24]. Other tools *reduce* the code to consider: Ge and colleagues' tool can detect and filter refactoring changes out of the review interface [25], whereas Tao and Kim's algorithm can slice a change into smaller, semantically coherent parts [26].

Comparing programs can also serve a educational purpose. Patitsas and colleagues' demonstrate gains in procedural knowledge when students view algorithms side-by-side rather than separately [27], although Price and colleagues' experiment suggests it also takes more time investment [28]. Outside the classroom, Glassman and colleagues' template-based EXAMPLORE, for example, finds uses of APIs and fits them into a common template, allowing users to see multiple examples of use in the same terms [29]. Other tools elucidate large samples of similar examples as trees, such as Cottrell and colleagues' GUIDO [30] or Middleton and Murphy-Hill's PERQUIMANS [31]. Meanwhile, Martie and colleagues' experiments with iterative, similarity-aware code search allows developers to discover reusable code through the gradual refinement of its features [32]. The effort of this research is to understand, refine, and motivate comparison tools like these.

III. STUDY

We focus on *comparative code interfaces*: webpages on which developers manage alternative programs. While the objects that are rendered visibly on a webpage (we call *features*) may be the most obvious content of a website, we are also interested in how those features are placed in relation to each other (*arrangements*), as in proximity or sequence. Content is not presented in a vacuum but enriched by its embodiment [33].

Our guiding research questions are as follows:

RQ1 How can interface *features* help developers navigate and compare similar snippets?

RQ2 How can interface *arrangements* help developers navigate and compare similar snippets?

A. Selected Contexts

We use the word *context* as shorthand for the interface and its corresponding expectations. Inspired by prior work on code comparison [7], we identified three contexts for which the existence of multiple similar code snippets is an essential feature of that context's purpose:

- **Search engines**, both code-centric like GitHub Search [34] and general like Google [35], where developers can sift through multiple webpages for a solution.

- **Question & answer platforms**, such as Stack Overflow [36], where developers can recommend code as an answer to another developer's problem.
- **Programming practice websites**, such as Leetcode [37] or CodeWars [38], where developers can compare how they solved a problem to other developers' programs.

B. Method: Initiation of Participatory Design

To generate new designs, we adopt techniques from a methodology called Participatory Design (PD). As Spinuzzi explains [39], PD emphasizes the *tacit knowledge* that practitioners apply without explicit articulation. Rather than design tools from theory and impose them upon its intended users, researchers observe work processes to prototype and iteratively refine tools *with* explicit collaboration from the users.

Therefore, PD is useful to software engineering and human-computer interaction researchers, whether as an end-to-end process or as a initial exploration [40], [41], [42]. For example, Johnson and colleagues use elements of a PD approach to learn how developers prefer static analysis tools [43], and Gorski and colleagues apply it within focus groups to sketch better warnings for cryptographic APIs [44]. These studies show how user-generated designs can bear dense insight in not only *what* something does but *how* it materially accomplishes it.

C. Recruitment

To recruit participants, we combined targeted recruitment—at large companies like Meta, Alphabet, and Microsoft, along with smaller firms too—and social media—professional associates on LinkedIn. Our recruitment message promised $50 in compensation for an hour of interaction. Potential participants filled in a Google Form with consent information and privacy rights, followed by three sections of information collection: years of experience, demography, and frequency with which they use our selected contexts. We used participants' responses to the last set to assign task contexts in our study that corresponds to their actual experience; for example, if a participant responded that they never use programming practice websites, we would not design in that context. Through these methods, we recruited the 11 developers in Table I.

D. Session Protocol

Our co-design activity for Zoom, teleconferencing software which transcribes, and Miro [45], a tool for design collaboration, consists of four phases distributed over an hour:

1) Introduce the activity and interface: We reiterate the general intent and content of our research, and the participant recreates a set of geometric shapes and textual notes using only the tools on Miro. We set no time limit for this activity, but in practice, no participant took longer than 10 minutes.

2-3) Discuss issues of current interfaces and design an ideal interface (two times): Each design task centers around a separate canvas ornamented with a textual description of the context, a note with reminders for the design task, and between six to ten moveable code snippets from a real but arbitrary example of that context, as shown in Figure 1. Before they designed anything, we asked participants about their experiences using existing interfaces, focusing on their *goals* and the *barriers* that get in the way. We captured themes of our discussion on sticky notes in the interface. After at least five minutes in discussion, we used the notes and code samples to start sketching the *speculative* interfaces (in contrast to recreating *current* interfaces). We encouraged them to think aloud and we asked follow-up questions to elaborate their statements. For moments when the design process slowed, we prepared topical questions to encourage reflection: the *priority* and *orientation* of alternatives, as well as *supplements to* or *transformation of* alternatives. We assigned two contexts to each participant (Table I) and spent about 30 minutes on the first task to account for the time of task understanding, versus 20 minutes on the second.

4) Generalize: After the tasks, we asked the participants to reflect on design principles common to their designs.

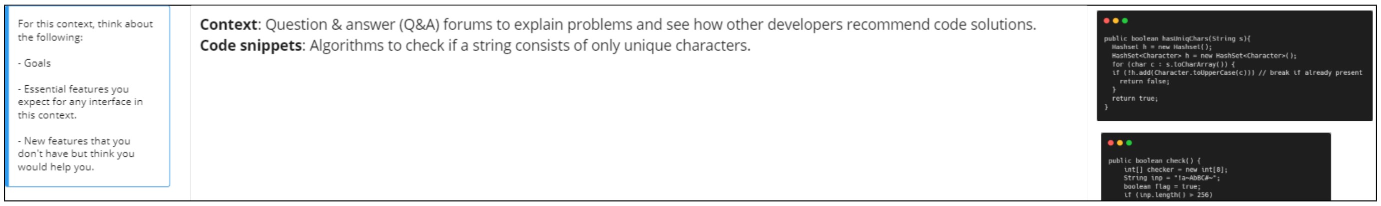
E. Analysis

Our raw data include 1) the interview transcripts, and 2) the visual designs. The first author cleaned each Zoom-generated transcript, divided it into each phase, separated long dialogues into distinct statements, and used an open coding scheme to create an initial set of codes [46] from explicit design choices—if the participant designed a visible search bar on the page, we considered "Search Bar"-related code. A statement could have any number of codes, including none, and we adjusted the coding approach by the context: asking about search interfaces presumes some *search functionality* whereas programming practice websites or Q&A platform contexts may not.

With transcripts and initial codes, the first two authors independently iterated over the transcripts again to refine the code-book and negotiate appropriate labels. After coding the first three sessions and negotiating and resolving disagreements, we transitioned to a randomized order of coding: all statements from the other eight sessions were aggregated and randomly collected into 15 sets, with small amounts of context on either side of a participant statement. We coded eight of these 15 sets in the same way as before: independent assignment before negotiated agreement and codebook refinement. For the final seven, we froze the codebook and relegated unrepresented concepts to a general "Other" category. Lastly, we organized our first-order codes into categories of the users' overall process. Cohen's Kappa throughout this process ranged between 0.38 and 0.63, averaging at moderate agreement [47]. We reflect on why reaching consistent agreement was difficult in Section VII.

IV. Findings

This section presents our emergent qualitative codes alongside evidence drawn from participant quotes. The higher-level categories that emerged, which we define more thoroughly in the next section include *Innate Code Traits*, *Contextual Code Traits*, *Interactions*, and *Coordinations*. We also had *Qualitative Modifiers*, a meta-code which refers to how we interpret the codes otherwise. Significant modifiers included whether a description was merely a description of current



For this context, think about the following:

- Goals

- Essential features you expect for any interface in this context.

- New features that you don't have but think you would help you.

Context: Question & answer (Q&A) forums to explain problems and see how other developers recommend code solutions.
Code snippets: Algorithms to check if a string consists of only unique characters.

```
public boolean hasUniqChars(String s){
  Hashset h = new Hashset();
  HashSet<Character> h = new HashSet<Character>();
  for (char c : s.toCharArray()) {
  if (!h.add(Character.toUpperCase(c))) // break if already present
    return false;
  }
  return true;
}
```

```
public boolean check() {
    int[] checker = new int[8];
    String inp = "!a~AbBC#~";
    boolean flag = true;
    if (inp.length() > 256)
```

Fig. 1. A cropped segment of the Miro interface in which the interfaces were sketched.

TABLE I
PARTICIPANTS IN THE DESIGN STUDY

Participant	Position	Years Exp.			Languages	Assignments		
		Programming	Prof. Programming	Prof. Designing		Learning	Searching	Rec.
P01	Grad. Student	5	2		Py., Java	✓		✓
P02	Grad. Student	6	1	1	Py., Java	✓	✓	
P03	Grad. Student	6	2		Py., Java		✓	✓
P04	Grad. Student	11	5	4	Py., Java	✓	✓	
P05	Grad. Student	9	2		Py., J.S.	✓		✓
P06	Software Eng.	12	7		Py., Java		✓	✓
P07	Software Eng.	5	1		Py., Java	✓		✓
P08	Data Eng.	25	23	15	J.S., C#		✓	✓
P09	Software Eng.	11	7		Py., Java	✓		✓
P10	Data Scientist	10	5		Py.		✓	✓
P11	Software Eng.	2	1		J.S.	✓	✓	

TABLE II
TOP 15 CO-OCCURRING CODES. '#P': # UNIQUE PARTICIPANTS, '#I': # UNIQUE INTERFACES, 'ALTS.': ALTERNATIVES

Code Pair		#P	#I
Ranking of Alts.	Active Crowd Eval.	10	14
Ranking of Alts.	Program Efficiency	6	7
Ranking of Alts.	Readability	5	7
Ranking of Alts.	Show Me Only #N Alts.	5	6
Ranking of Alts.	Filter / Search for Alts.	4	6
Filter / Search for Alts.	Language	6	6
Filter / Search for Alts.	Relevant Motivating Context	6	6
Filter / Search for Alts.	Libraries and Dependencies	4	6
Filter / Search for Alts.	Tags & Keywords	4	5
Program Efficiency	Readability	5	7
Program Efficiency	Automatic Comparator	4	5
Program Efficiency	Language	4	4
Program Efficiency	Code Documentation	4	4
Show Me Only #N Alts.	Vertically-Oriented Alts.	5	6
Embedded IDE	Libraries and Dependencies	5	6

processes, an identification of a barrier, or the negation of a feature in that a participant does *not* want it. Table III lists all codes that did not appear with modifiers, Table IV lists the top 10 codes that co-occurred with our IDENTIFYING A BARRIER code, and Table II lists top co-occurring codes otherwise.

We present each quote with a reference such as P01$_L$, indicating the participant who said it (P01) and the type of interface they were designing during the quote (S: Searching, R: Recommending, L: Learning, and G: General Reflections). In the text, qualitative codes will be in small caps and will be accompanied in section headings by the number of participants who discussed it in one of their contexts (e.g. LANGUAGE (10P.)).

A. RQ1: Features for Comparative Interfaces

Under "features," we include information implied in a single alternative (*innate* code features) and information that emerges by virtue of its presence on a multimedia website (*contextual*). For example, any given algorithm would have the same computational complexity regardless of which website hosts it, but the types of documentation provided for that algorithm could change with each upload.

1) *Innate Code Traits*:

LANGUAGE *(10P.)*: Developers often want code only in their language of active work or practice. P04$_L$ clarifies that language is essential for comparing in common terms: "*any comparison between language does not make sense because every language has a completely different system.*" As such, the codes LANGUAGE and FILTER / SEARCH are one of those most common code pairs in Table II However, when the goal is learning a new language, P08$_R$ permits for comparison across languages: "*I got to start learning Python serious for data ingestion. I've done C# for a long time, and so I'm coming from there, I would like to compare them and see.*" In their case, controlling for language allows them to position familiar languages with the unfamiliar.

READABILITY *(10P.)*: Readability not only allows developers to evaluate individual alternatives, but it also acts as a heuristic for credibility among alternatives. P11$_S$ states, "*I almost do not trust their code as much because they're not practicing clean writing techniques.*" As such, poor readability emerges as the most common barrier in Table IV. Nevertheless, for many, readability outweighs high-performing but obscure coding tricks. In a recommendation context, P06$_R$ indicates that "*cleverness*" might be performant but can "*introduce unnecessary complexity that makes it hard for the team to intake.*" As P03$_G$ reflects overall, "*if I'm comparing two code snippets, it's how readable it is, so comments, indentation; and how efficient it is is secondary.*"

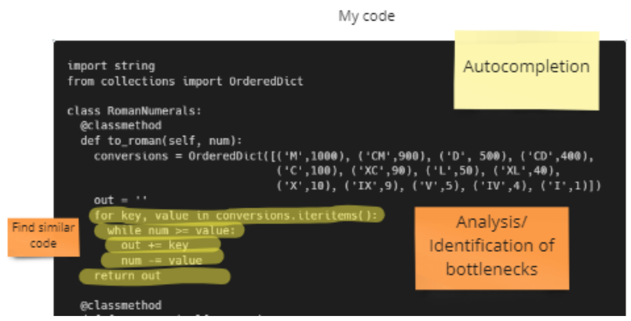
PROGRAM EFFICIENCY *(9P.)*: Performance analyses, when explicitly included as either static features or dynamic generation, help developers comprehend another dimension of code both before and after they understand the syntax. P01$_R$ frames this as a reaction to the readability barrier: "*code is not always very clear and concise to read, so if I can get these kinds of statistics, that will also help tremendously.*" If present, as often is the case for learning contexts, these analyses could fit into their workflow of evaluating each answer—"*You can do your own internal evaluation of the logic, and then your eyes can*

TABLE III
Qualitative codes. '#P': number of unique participants (out of 11), '#I': the number of unique interfaces (out of 22). 'S|R|L': number of unique Searching, Recommending, and Learning contexts in which this code was discussed respectively. Codes that occurred < four times in both #P and #I are omitted.

	Code	Definition	#P	#I	S	R	L
Innate	Language	In what programming language the alternative is written.	10	14	5	4	5
	Readability	How the code is structured for ease of human reading	10	13	3	6	4
	Program Efficiency	An algorithm's runtime or space complexity given the problem.	9	13	2	5	6
	Libraries and Dependencies	How many imported libraries or features an alternative requires.	8	12	5	5	2
	Program Structure	The syntactic components of an algorithm.	6	6	0	2	4
	Code Length	How long in lines or characters an alternative is.	5	5	1	0	4
Contextual	Relevant Motivating Context	The natural language question to which all alternatives relate.	11	19	6	7	6
	Code Documentation	Natural language explanations of design or behavior.	11	18	5	7	6
	Active Crowd Evaluation	Quantitative userbase feedback (e.g. voting)	11	16	4	7	5
	Crowd Discussion	Natural language userbase conversations about an alternative	8	11	2	6	3
	Tags & Keywords	Short, explicitly visible labels that describe an element of the alternative.	7	8	2	5	1
	Signals of Source Credibility	Indications of the author's or platform's expertise in the topic.	7	7	2	4	1
	Related Topics & Suggestions	Links to other pages on the platform that are related to the current one.	6	8	4	3	1
	Verified Solutions	Indications that an alternative has been manually, credibly confirmed.	6	7	1	3	3
	Views & Frequency	Indications of other users passively interacting with an alternative.	5	5	3	0	2
	Recency of Alternative	When the alternative was written or valid.	3	4	2	1	1
Interaction	Ranking of Alternatives	Prioritization of alternatives based on selected traits.	11	20	6	7	7
	Automatic Code Formatter	Consistent, superficial modification of an alternative's syntax.	10	13	3	7	3
	Filter / Search for Alternatives	Removal or inclusion of alternatives based on selected traits.	9	13	6	5	2
	Embedded IDE	In-place editability and executability of alternatives by the user.	9	12	2	6	4
	Automatic Comparator	In-place analysis of similarities and differences between alternatives.	9	9	1	4	4
	Error Detection	Automatic detection of issues with syntax or tested behavior.	7	10	1	4	5
	Easy Extraction of Alternatives	Assistance in copying-and-pasting an alternative and dependencies.	7	8	3	4	1
	Automatic Refactoring	Altering the tokens of code according to predefined rules.	5	5	0	3	2
Coordination	Vertically-Oriented Alternatives	Alternatives are placed top to bottom (*must be explicitly mentioned*)	11	15	3	8	4
	Horizontally-Oriented Alternatives	Alternatives are placed side to side (*must be explicitly mentioned*)	11	14	3	5	6
	Show Me Only #N Alternatives	Limits on how many alternatives should be visible at once.	10	13	4	4	5
	Hyperlinks / Click to Alternatives	Alternatives are located on separate pages linked to from this page.	10	13	4	3	6
	Consider Diversity of Alternatives	Consideration of the (dis)similarity of alternatives in positioning.	8	8	1	2	5
	Dynamic Refocusing	Reconfigurability of which alternatives are visible on the page.	6	9	3	2	4
	Information on Hover	Features that are visible only when another feature is in focus.	6	7	3	2	2
	Partially Revealed Alternatives	Partial visibility of alternatives until specifically selected.	4	3	2	0	1

TABLE IV
Top 11 codes co-occurring with the Barrier code. '#P': # unique participants, '#I': # unique interfaces, 'Alts.': Alternatives

Barrier	**#P**	#I	**Example**
Readability	7	8	P02$_R$: "*The top solutions are sometimes too complex and they don't have any comments within them.*"
Filter / Search for Alts.	6	7	P06$_R$: "*One of the big obstacles, being able to phrase my question in a way that makes or in a way that actually surfaces useful answers.*"
Libraries and Dependencies	4	5	P11$_S$: "*They have the same issue, but it's not going to solve my issue because they're not using [dependencies] X, Y, and Z, they're only using X and Y.*"
Program Efficiency	4	5	P05$_L$: "*That histogram [of performance], which is in my experience horribly inaccurate.*"
Relevant Motivating Context	4	4	P02$_S$: "*The biggest problem with those would be, this is taking input in a different way than I want and disseminating output in a different way.*"
Crowd Discussion	4	4	P08$_R$: "*People who answer questions just to try to look smart...Sometimes just being pompous*"
Embedded IDE	4	4	P7$_R$: "*If you implement something like this, it would be a really slow experience. Say you have hundreds of people, you cannot essentially have an entire embedded editor for each and every answer.*"
Show Me Only #N Alts.	4	4	P11$_L$: "*If you have too many winners, I feel like that would be somewhat overwhelming.*"
Easy Extraction of Alts.	3	3	P07$_R$: "*Most of the times it is unable to pick up the code indentation, to preserve it.*"
Code Documentation	3	3	P07$_G$: "*They should have some comments or description about that particular snippet just to make understanding [easier].*"
Hyperlinks / Click to Alts.	3	3	P07$_G$: "*Anything that's more than three clicks is a bad experience.*"



```
import string
from collections import OrderedDict

class RomanNumerals:
  @classmethod
  def to_roman(self, num):
    conversions = OrderedDict([('M',1000), ('CM',900), ('D', 500), ('CD',400),
                               ('C',100), ('XC',90), ('L',50), ('XL',40),
                               ('X',10), ('IX',9), ('V',5), ('IV',4), ('I',1)])
    out = ''
    for key, value in conversions.iteritems():
      while num >= value:
        out += key
        num -= value
    return out

  @classmethod
```

Fig. 2. P04 designs a learning interface that can identify potential problem areas and links to how other developers solved those same problems.

gradually move over into the grading of it" (P09$_R$).

In a speculative design mode, P04$_L$ takes the concept further; rather than profiling the algorithm as one block, a tool that breaks the program down and understands performance at a smaller granularity could help them optimize. Figure 2 shows their sketch of a learning interface that combines efficiency and structural analyses to highlight the most problematic areas:

> So I could click at this highlighted section, and it can actually just scroll to a place where it would show this [other algorithm] is doing it better, this is doing the same thing I tried to do, because there could be a completely different approach.

However, performance analyses are only practical in certain circumstances. P05$_L$ raises concerns about fixating on small optimizations—"*If I go about trying to get the top 10% fastest solutions, that is going to take me, I don't know, three, four hours; I wonder if that effort is worth it*"—to say nothing of the analyses' accuracy ("*in my experience, horribly inaccurate*") In search or recommendation contexts, these features are not common because "*the onus is too great*" for individual participants to provide themselves (P09$_R$) and developer's different working environments pose challenges to any centralized measure of efficiency on the web (P03$_S$).

***Other Innate Code Traits**:* PROGRAM STRUCTURE (6P.) focuses on text of algorithms apart from their behavioral efficiency, such as number of variables or data structures. P11$_L$ discusses a speculative application of searching algorithms by their use of code features—"*if you could search for things this code was solved without; a for-loop [for example], how did they do that?*" Likewise, CODE LENGTH (5P.) is related to READABILITY but refers only to textual size. Smaller algorithms are preferred until they become inscrutable with unusual shortcuts.

2) *Contextual Code Traits:*

***RELEVANT MOTIVATING CONTEXT** (11P.) & **RELATED TOPICS & SUGGESTIONS** (6P.):* CONTEXT is not just a program specification but the greater story of *why you care*. On search engines, the relevant context may be the query; on recommendation websites, the instigating question; and on learning platforms, the programmatic challenge. The barriers that emerge around this code relate to whether your context matches the resource's—"*sometimes the questions are too specific on these questions platform to the stack overflow. and I'd like some curation that generalizes problems*" (P05$_R$). Furthermore, the interface can expedite the search for the right context by linking to similar contexts nearby: "*maybe that particular question doesn't answer my query completely; possibly related questions or tags to related questions [would help]*" (P03$_R$). RELATED TOPICS & SUGGESTIONS, including links to similar pages and suggestions within search engines, allows for agility between contexts.

***DOCUMENTATION** (11P.) & **CROWD DISCUSSION** (8P.):* Unlike a CONTEXT which applies to all alternatives on a page, DOCUMENTATION explains for one specific alternative. Having documentation on appropriate use cases can filter out irrelevant code apart than manual code investigation:

> (P06$_R$) A lot of those solutions may be more useful for particular inputs or outputs...In that case, comparing individual, like, one sort algorithm against another may not be quite as useful as being able to find the one that matches your use case the best.

In other words, comprehension through good documentation can be a strategy to narrow the field of reasonable comparison.

A related code was CROWD DISCUSSION, where other developers can leave comments for the author. These are valuable for flagging possible issues with an algorithm and so are valuable for refining alternatives indefinitely. However, four developers raise issues they've encountered with those environments, such as the unhelpful attitudes of participants.

***ACTIVE CROWD EVALUATION** (11P.), **VIEWS & FREQUENCY** (5P.), & **VERIFIED SOLUTIONS** (6P.):* Developers trust the crowd on matters of usefulness, so feedback is a quality gate before comprehension effort. Quality signals, however, are not comprehension itself, as P09$_R$ says: "*[comprehension] is left up to, more or less, my experience as a developer and how well the person explained the rationale.*" Still, as P01$_R$ says, voting lets them jump to reasonable conclusions: "*if the interface does not have any votes or nothing, then I don't know if this answer is actually right or wrong.*"

These signals are often not available via search engines directly—as P11$_G$ says, "*not everyone is searching my exact search*"—although a metric of general popularity of the resource may be considered in ranking. VIEWS & FREQUENCY represents interactions passively, putting the onus on the platform over users to measure popularity.

While the crowd has its use, developers value authoritative recognition in the form of VERIFIED SOLUTIONS. For example, P05$_R$ looks for the Stack Overflow's green checkmark, indicating the author's preferred answer. When time is short, this recognition precludes any comparison at all, as P06$_R$ finds that "*a lot of times, I'm probably not comparing so much as just choosing or spending most of my time focused on what someone else has chosen as the correct answer to the question.*"

***Other Contextual Code Traits**:* TAGS & KEYWORDS (7P.) summarize themes into categories you can manipulate. SIGNALS of Source Credibility (7p.) include individual authors or websites that developers trust to give quality answers, compared to unknown or disreputable sources. Recency of Alternative (3p.) deals with when a program was valid. This code reflects the propensity of languages and libraries to evolve over time.

B. RQ2: Arrangements for Comparative Interfaces

In this section, we focus on the dynamism and physical space of an interface, focusing on code-centric features that change the environment (*interactions*) or describe algorithms using visible relationships (*coordinations)*.

1) *Interactions*:

Filter / Search for *(9p.)* & Rank Alternatives *(11p.)*: Full comprehension takes time, too much so to evaluate each snippet individually. The search bar and filter narrow the field from global to specific, whereas the ranking algorithm prioritizes among a given set. All three are among the core tools available across interfaces that relocate the onus of manual iterations from developer to platform, and many of the code traits in the previous section are candidates for input. The barriers that emerge are akin to those we discussed for motivating context but more in articulating needs instead of strictly matching them (P06$_R$: "*Being able to phrase my question in a way that actually surfaces useful answers*").

Embedded IDE *(9p.)* & Easy Extraction of Alternatives *(7p.)*: Comprehension comes not only from reading code but also experimentation and tinkering. For P08$_R$, the miniature IDE is "*the biggest leap forward*" for working with code online in the past few years, and adds that it would be useful to select specific versions of languages and libraries. In the absence of an in-place IDE, quick transitions from context to IDE can also be accomplished through facilitated copying-and-pasting, or what we call the Easy Extraction of Alternatives. P03$_R$ describes "*a button with the code snippets which you have which allows you to copy the code directly*" from its original context into your IDE. Beyond ease in transplanting (proper attributions notwithstanding), this code also includes making snippets self-sufficient through the explicit inclusion of the proper libraries, data structures, or formatting.

Automatic Code Formatter *(10p.)* & Automatic Refactoring *(5p.)*: Just as innate readability is an important code trait, developers say they would benefit from automatic readability improvements. This can be as simple as displaying code uniquely on the page so that it stands out among other features—"*at least some styling which will differentiate the comments the loops and the functions*" (P02$_S$). On the other, it may be transforming the spacing—"*some sort of feature that would automatically indent these, and line it up the right way with the spacing*" (P11$_S$). For comparison, normalizing the format of alternatives may eliminate non-functional differences and clarify the differing structure— "*It'd be useful to be able to remove those dissimilarities [tabs versus spaces, quotes, double-quotes] and focus on, specifically the the solution*" (P06$_S$)

Beyond format, developers may benefit when formatting separates discrete tasks within the algorithm. At this point, the line between a Formatter and a Refactoring tool becomes fuzzy, and participants describe more in-depth re-writing of code. P01$_R$ speculates on assistance to novices especially:

> We can add a button which simplifies code and if you click that button, the code is broken down into single statements: one line, one statement. For example...in Python, we have a list comprehension; sometimes, it's difficult to understand for beginners.

Automatic Comparator *(9p.)*: A comparator is any tool that supports the intentional juxtaposition of programs and its traits. We find two ways that a comparative can operate: as the *analytical engine* that ingests alternatives and produces new information, or the *scaffold* which takes what is known and expedites its discovery by perceptually emphasizing its contrast. For example, ranking algorithms are an application of comparison over indefinitely large sets by ordinally measuring algorithms on some trait (the analysis) and arranging in order (the scaffold). However, ranking is less clear for measurements that are not straightforwardly orderable, or more complex combinations of metrics (P02$_R$: "*number of variables, this code runs in this time, this code takes this much space compared to your code,...maybe number of lines in the code*").

Nevertheless, innate and contextual traits can be dimensions for the comparator, and they could scaffold this information:

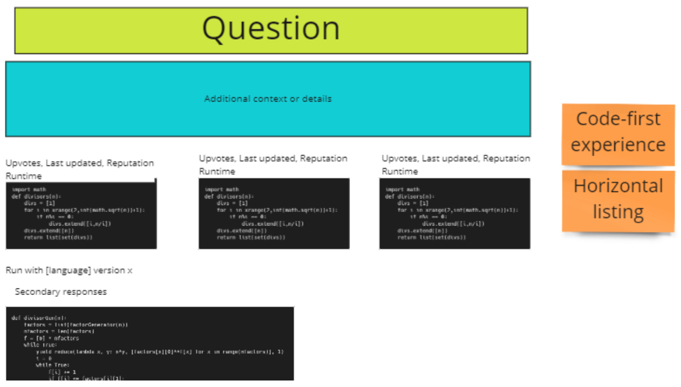
> (P09$_R$) If there was some way to grade snippets you know, using the same objective rubric or standard or something, then that would be really interesting and could provide meaningful ways to differentiate which one of these may suit your needs better if they both solve the problem.

However, P10$_R$ brings up a complex constraint: meaningful differentiation "*requires things to be structured or similar way.*" They leave it unresolved whether structural consistency should be the user's or context's responsibility. This concept in particular spurred many statements about technological challenge: "*neat but would be challenging,*" in P06$_S$'s estimation.

2) *Coordinations*:

Horizontally- *(11p.)* and Vertically-Oriented Alternatives *(11p.)*: Verticality is assessed as the more comfortable arrangement for rapidly scrolling through multiple alternatives. With most screens extending up and down, it easy to embrace "*the infinite scroll, you can just scroll with your fingers*" (P05$_G$). P01$_R$ says it reduces clicks when navigating large fields, "*because if we have something go right or go left, you click on a button to move to the next code, that's too many clicks.*" More simply, P10$_S$ says this is what most people are used to ("*Google's probably just trained me very well*").

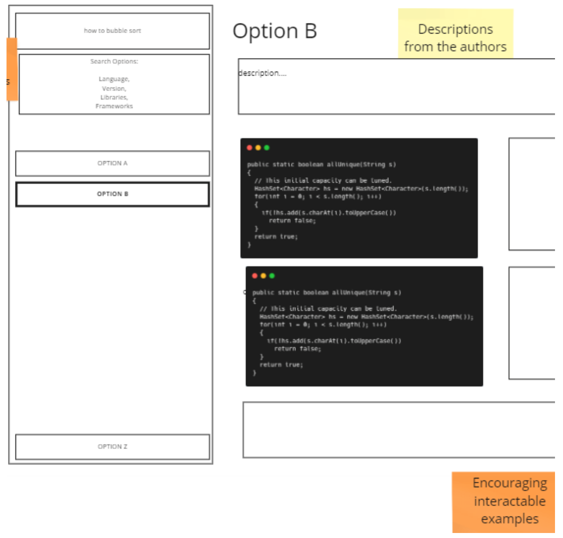
However, horizontality may be an acceptable alternative when the goal is intentional comparison between smaller selections. Participants use other metaphors to describe its value: P09$_R$ says their interface could "*condense it horizontally, like a book would*", and P11$_L$ clarifies for theirs, "*it reminds me of a coding screen when I do my work: I have one screen looking*


Question
Additional context or details
Upvotes, Last updated, Reputation Runtime
Upvotes, Last updated, Reputation Runtime
Upvotes, Last updated, Reputation Runtime
Run with [language] version x
Secondary responses
Code-first experience
Horizontal listing

```
import math
def divisors(n):
    divs = [1]
    for i in xrange(2,int(math.sqrt(n))+1):
        if n%i == 0:
            divs.extend([i,n/i])
    divs.extend([n])
    return list(set(divs))
```

```
def divisorGen(n):
    factors = list(factorGenerator(n))
    nfactors = len(factors)
    f = [0] * nfactors
    while True:
        yield reduce(lambda x, y: x*y, [factors[x][0]**f[x] for x in range(nfactors)], 1)
        i = 0
        while True:
            f[i] += 1
            if f[i] <= factors[i][1]:
```

Fig. 3. P10's recommendation interface explores horizontal orientations for the top results and vertical for the rest.



```
public static boolean allUnique(String s)
{
  // This initial capacity can be tuned.
  HashSet<Character> hs = new HashSet<Character>(s.length());
  for(int i = 0; i < s.length(); i++)
  {
    if(!hs.add(s.charAt(i).toUpperCase())
      return false;
  }
  return true;
}
```

```
public static boolean allUnique(String s)
{
  // This initial capacity can be tuned.
  HashSet<Character> hs = new HashSet<Character>(s.length());
  for(int i = 0; i < s.length(); i++)
  {
    if(!hs.add(s.charAt(i).toUpperCase())
      return false;
  }
  return true;
}
```

Fig. 4. In P06's recommendation interface, a number of hyperlinks with brief descriptions provide quick access to a variety of solutions based on their query.

at my code and then I have another screen looking at the issue." However, P04$_L$ says the value of horizontal alignment is also a function of structural similarity: "*When someone has written something very similar and just the structure is a little different then it's just easier for me to compare side by side.*"

Because of their different perceived benefits, there may be value in supporting both vertical and horizontal capabilities. In recommendation contexts, P10$_R$ mixes them, as shown in Figure 3. On the other hand, some participants think that focusing on specific orientation may be missing the point. As P06$_R$ says, "*I don't think there's much of a difference, side by side versus up and down, and I think just having them close together likely will make them easier to compare now.*" Instead, the emphasis may be on pure proximity (P02$_S$: "*it's really good to see both codes in the same screen.*")

Show Me Only #N Alternatives *(10p.)*, Dynamic Refocusing *(6p.)*, & Consider Diversity of Alternatives *(8p.)*: Participants want good curation, not infinite choices. A repetitive theme is to prefer a few, often two or three of the highest ranked (P11$_L$: "*shortest, the one with the most votes, the easiest to read*"). Search results can grow longer, but developers may look at only "*the first half dozen results, and then I'm probably more likely to try and tune my search query*" (P06$_S$).

Especially within the learning context, there is the concern for the variety of ways a problem can be solved. Consider Diversity of Alternatives refers to the benefits in conceptual awareness that may come when alternatives take different approaches rather minor variations of the same structural approach (e.g. names and syntactic sugar). P09$_L$ explains:

> You could have some kind of grading system and just maybe keep the top three unique solutions, because otherwise you know you could just look at the first place solution, copy it and paste it and now you're the second place winner.

Furthermore, automatic ranking may not always match developer preference. Dynamic Refocusing is the ability to manually rearrange the alternatives so that the interface emphasizes *your* choice instead of theirs. Some participants effectively accomplish this through tabbed web browsers or IDEs, but P04$_S$ speculates about integrated tools that allow you to "*drag and drop [two examples] to the side*" as though you're "*activating solutions.*" P06$_R$ uses shopping metaphors and builds on the concept of automatic comparators:

> Maybe there's a way also to have multiple snippets here, for the user to hide one or bring two of them closer together. A lot of sites allow you to like choose two or three different products and then compare them side by side and see a list of features.

Hyperlinks / Click to Alternatives *(10p.)*, Information on Hover *(6p.)*, & Partially Revealed Alternatives *(4p.)*: Developers demonstrated a variety of options to reduce the actual visible code while maintaining access to them. Descriptive hyperlinks are a space-conscious way to indicate more candidates elsewhere at the cost of context switching. P06$_R$'s interface in Figure 4 organizes relevant content through buttons that activate dynamic content, allowing them to reduce the number of snippets on screen at once too.

However, there is a cost to hyperlinks if you need to backtrack through numerous webpages. As P07$_G$ says, "*anything that's more than three clicks is a bad experience*". As such, participants propose two implementations that compromise for space and effort. The first is Information on Hovering, where a tooltip appears temporarily when the cursor is over specific features, presenting relevant information only when requested. For P09$_L$, this preserves momentum while learning many new concepts "*because otherwise you're just going to have to go Google stuff and that'll take you away to a different web page entirely.*" The second approach is to provide truncated snippets of a larger algorithm, or partially revealed alternatives. This happens in some search engines, both code-specific (i.e. GitHub code search) and general (i.e. DuckDuckGo). This design may enable a quick end to the search process if the window contains all the code features necessary for the developers' goal. As P06$_S$ describes it, "*I don't even navigate to the page, I can copy it directly off of the search result view in the description.*" P11$_S$ describes an approach that combines ranking with the size of the revealed portion: "*Usually my top ones are the best options, so I'd rather have bigger code snippets underneath them then smaller ones with more description then more options with less description.*"

V. Discussion

The findings describe features that developers believe will help them with comprehension and comparison. In this section, we relate their designs to the broader research field.

A. A Fourfold Model of Contextual Comparison

Whereas we organized our codes by how information is embodied on the page, an alternative organization is by *how a feature or arrangement supports comparison*. From this frame, we envision four behaviors within web-powered comparison:

- *Global Evaluation*: The developer navigates between all possible contexts with single or multiple alternatives via SEARCH, LINKS, RELATED TOPICS, and similar features.
- *Local Evaluation*: The developer navigates among alternatives in a single context with the help of FILTERS and skimmable assessments of individual alternatives.
- *Individual Comprehension*: The developer deepens their understanding of a single alternative with the help of DOCUMENTATION, DISCUSSION, and interactive tools.
- *Comparative Comprehension*: The developer assesses multiple alternatives using the same terms or questions, consulting the RANKING or any COMPARATOR-like tools.

The variety of algorithmic signals resonates with *Information Foraging Theory* (IFT), a framework proposed by Pirolli & Card as an adaptation of behavioral ecology for information workers [48]. In this theory, the worker is a *predator* whose *prey* is some information feature. The predator traverses *patches* of features via *links*, and each link may be designed with *cues*, or features that suggest the linked content. This theory has been applied to software development tasks by Fleming, Scaffidi, Piorkowski and others [49], and expanded to include similarity in Ragathan [13], [8] and Kuttal [15]. Our work resonates with their efforts: if the context's webpage is a patch, then innate code traits characterize the prey, contextual code traits influence the scent, interactions enrich the patch, and the arrangements make traversal within a patch more effective.

B. To Complement or Complicate Comparative Comprehension

Previous work on code comparison in laboratory settings [7] focuses on what we call "innate code traits"—the language, the (un)readable text, the structural composition of algorithms—or lightweight comparators in the case of review [22]. Both deal with tool studies but in different contexts: there, within the IDE and review interfaces; here, within the online sources of algorithms that may later lead into the IDE. Nevertheless, the barriers to comprehension and comparison in all of these works have consonances across contexts as well as distinctions. Although this research does not quantify the inaccuracy of comparison as the other studies do, it does elaborate on the barriers to efficient work as well. These barriers include inherent difficulty of comprehending code as well as making connections between the gulf of large syntactic differences, yet also touch on issues "above" the code: abrasive social interactions, or insufficient or excessive information on a single page. In other words, these research efforts paint a picture of a stack of barriers running through a holistic software process.

Nevertheless, code comparison is one behavioral option in the economy of software comprehension. While choosing to compare can lead the developer into much more nuanced understandings of language and program behavior, it costs time and effort [28]. The notion of developers being pragmatic about the decisions of their software behavior as been explored in research likes Holmes and Walker's [18], as in Piorkowski's reuse of the concept of production bias, or "the tendency of software developers to view learning as a costly task" [50].

Therefore, the follow-up question is this: *what behaviors does explicit comparison extend or replace?* Even if only one alternative is considered, checking *satisfaction* is only possible if compared to the abstract requirements of the search in the first place. Granted, requirements-to-program comparison may not engage exactly the same skills as the program-to-program comparison discussed here [51]. Beyond this, *implicit* comparison iterates one-by-one over alternatives until a satisfactory alternative is found and choosing to look no further, albeit the memory of prior alternatives considered may influence choices from the background.

Lastly, automation can eliminate human element of memory altogether; the computational tool can make recommendations without revealing all of the considered options. This option, which we can call *delegated* comparison, can be envisioned as a trusted programming assistant. Still, a design choice remains: how much of the recommendation reasoning process will be made transparent and accountable? While tool automation has clear application for assisting decision making, delegating analysis up to *replacing* human involvement may obscure information that developers value [52].

C. Implications For Tools

This research can generate principles and directions which may prove productive for future tools, such as the following:

- *Flexible Environments for Multiple Alternatives:* Developers in this study describe manipulating the browsers' tools—tabs and windows—to accomplish spatial proximity for lack of specialized tooling to bring alternatives into the same view. Additionally, the act of tracking down alternatives also has actions with different costs—opening a new window, revealing hidden content, and so on. Therefore, tools for dynamically bringing alternatives into view, moving between orientations, and maintaining a history of alternatives considered may streamline the ergonomics of the comparative process.
- *Preserving Diversity in Multioptimization:* Developers have numerous priorities in "good" code—efficiency, maintainabil-

ity, and readability, for example—and any one taken too far can be deleterious for others. Additionally, developers are averse to being overwhelmed with alternatives. Therefore, privileging a few results along significant facets and offering iterative refinement [32] remains a promising path forward.

- *Separating the Superficial From the Semantic:* The style of code may constitute a difference without making a difference in the phenomena themselves. Aligning known features, both visually and behaviorally, with unfamiliar features may allow the transfer of conceptual knowledge. If navigating difference is costly and syntactic difference contributes to the cost, then tools to normalize structures and appearance can redirect developer attention to more fruitful endeavors.

VI. Future Work

Iterating Participation in the Design: Although the literature of PD inspired our methodology, the research described here fulfills only the first few steps of the full method. As Spinuzzi clarifies, "participants' cointerpretation of the research is not just confirmatory but an essential part of the process" [39]. The work performed here can serve as the first iteration in a larger research process. First, *confirm* with our participants that the themes we identified match their experience and *complicate* findings with new preferences that participants can identify only when face-to-face with the sum of their statements. Second, *prototype* interfaces based on the sketches here and elicit additional feedback to reveal incongruities in the designs. Third, *iterate* and *revisit* these methods in longer durations. These processes could occur with the same participants or new ones; both have tradeoffs in the depth and breadth of perspectives.

Understanding the Bespoke Comparative Environment: This study explores algorithms not just concepts but as physical artifacts within a developers' sensory perception. This concern goes beyond readability, the premise of which assumes that "same" algorithms can differ on the internal relationships of their parts. Rather, codes like VERTICALLY- and HORIZONTALLY-ORIENTED ALTERNATIVES deal with the external relationships between physical algorithms. While we had participants reflect from their concrete experiences and preferences, there remains insights to glean from direct observations of how developers currently shape their environments. Our results here suggest many interfaces currently support comparison only implicitly, not through tools, and developers reshape their environments opportunistically through windows and tabs to rearrange algorithms according to immediate needs. Codes like DYNAMIC REFOCUSING and SHOW ME ONLY # ALTERNATIVES demonstrate developer desire to shape the comparative environment according to preference, so researchers who directly observe this phenomenon could elaborate opportunities to facilitate it.

Generalizing to New Paradigms: It is worth noting that this study was performed before generative AI tools, such as ChatGPT, were in wide use among developers [6]. The presence of such tools provides yet another interface for developers to perform code comparison. Like search engines, and the other contexts described in this work, comparison is not a primary design consideration in the current generic generative AI interfaces, so findings in this work may generalize to those additional contexts while being transformed by the conversational approach.

VII. Threats

Internal to this study, participants may limit themselves based on their confidence or the pressure of observation. Furthermore, all participants spoke extemporaneously without time to reflect on subtler implications of their ideas. As evidenced by Table I, very few of our participants have professional design experience, and it is possible they could reevaluate initial assessments when seen implemented. To this, we reassured participants that, first, we were not judging them on their design quality, and second, we were not limiting their ideas to what was feasible with current and familiar technology. Future iterations of a participatory design approach would directly address this, if not even opening up the design considerations in this study to a wider audience with a survey.

Additionally, our qualitative coding of the interviews may also be imprecise. For example, the separation of long participant statements into smaller statements was performed by the first author's intuition about separate ideas and manageable statement lengths for the coders. These choices have direct consequences for the frequency and co-occurrence of codes. Therefore, we prefer to consider themes by how many participants or sessions that arose in uniquely, rather than the raw frequency of mentions.

Furthermore, the code-book came from the negotiation of two coders but struggled to reach a high agreement for some transcripts. Some of the biggest challenges to coding agreement included first, being attuned to all ideas in a statement given that we allowed multiple codes, and second, distinguishing when statements referred to participants' interface designs and not merely to the current, less-than-ideal situation. The use of multiple coders demonstrably improves the reliability of the results but nevertheless indicates room for improving the protocol in future iterations.

VIII. Conclusion

In this study, we used a collaborative design activity to probe for developers' implicit preferences for contextual code comparison. After eliciting their goals and barriers, we charged the developers to reimagine familiar web interfaces to better help the comparison of alternative reusable code snippets and making decisions from their contrasts. This methodology gives us insight into many of the qualities of code, both innate and contextual, that developers consider important for their tasks, as well as the opportunities for transforming or more productively moving through similar programs. The framework of these concepts can raise further inquiries into the mechanisms of comparative comprehension and inspire future approaches to code-centric designs.

Acknowledgements

This research is supported in part by NSF SHF #2006947 and NSF SHF #1749936.

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[2] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 191–201.

[3] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[4] S. Baltes and C. Treude, "Code duplication on stack overflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 13–16.

[5] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 291–301.

[6] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," *IEEE Transactions on Software Engineering*, 2024.

[7] J. Middleton and K. T. Stolee, "Understanding similar code through comparative comprehension," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2022, pp. 1–11.

[8] S. S. Ragavan, B. Pandya, D. Piorkowski, C. Hill, S. K. Kuttal, A. Sarma, and M. Burnett, "Pfis-v: modeling foraging behavior in the presence of variants," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, 2017, pp. 6232–6244.

[9] M.-A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, 2005, pp. 181–191.

[10] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

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

[12] F. Détienne, "Reasoning from a schema and from an analog in software code reuse," *arXiv preprint cs/0701200*, 2007.

[13] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, "Foraging among an overabundance of similar variants," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 3509–3521.

[14] G. R. Bai, J. Kayani, and K. T. Stolee, "How graduate computing students search when using an unfamiliar programming language," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 160–171.

[15] S. K. Kuttal, S. Y. Kim, C. Martos, and A. Bejarano, "How end-user programmers forage in online repositories? an information foraging perspective," *Journal of Computer Languages*, vol. 62, p. 101010, 2021.

[16] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?(nier track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 804–807.

[17] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, vol. 24, pp. 637–673, 2019.

[18] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–44, 2013.

[19] E. W. Myers, "An o (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[20] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[21] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Confusion in code reviews: Reasons, impacts, and coping strategies," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 49–60.

[22] J.-P. O. Middleton, Justin and K. T. Stolee, "Barriers for students during code change comprehension," in *2024 IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2024.

[23] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 679–690.

[24] K. Etemadi, A. Sharma, F. Madeiral, and M. Monperrus, "Augmenting diffs with runtime information," *IEEE Transactions on Software Engineering*, 2023.

[25] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill, "Refactoring-aware code review," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 71–79.

[26] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 180–190.

[27] E. Patitsas, M. Craig, and S. Easterbrook, "Comparing and contrasting different algorithms leads to increased student learning," in *Proceedings of the ninth annual international ACM conference on International computing education research*, 2013, pp. 145–152.

[28] T. W. Price, J. J. Williams, J. Solyst, and S. Marwan, "Engaging students with instructor solutions in online programming homework," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–7.

[29] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing api usage examples at scale," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.

[30] R. Cottrell, B. Goyette, R. Holmes, R. J. Walker, and J. Denzinger, "Compare and contrast: Visual exploration of source code examples," in *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2009, pp. 29–32.

[31] J. Middleton and E. Murphy-Hill, "Perquimans: a tool for visualizing patterns of spreadsheet function combinations," in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2016, pp. 51–60.

[32] L. Martie, A. v. d. Hoek, and T. Kwak, "Understanding the impact of support for iteration on code search," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 774–785.

[33] M. Matera, F. Rizzo, and G. T. Carughi, "Web usability: Principles and evaluation methods," *Web engineering*, pp. 143–180, 2006.

[34] "Search · GitHub," https://github.com/search, Last accessed 2024-07-07.

[35] "Google," https://www.google.com/, Last accessed 2024-07-07.

[36] "Stack Overflow," https://stackoverflow.com/, Last accessed 2024-07-07.

[37] "LeetCode," https://leetcode.com/, Last accessed 2024-07-07.

[38] "Codewars," https://www.codewars.com/, Last accessed 2024-07-07.

[39] C. Spinuzzi, "The methodology of participatory design," *Technical communication*, vol. 52, no. 2, pp. 163–174, 2005.

[40] K. Kautz, "Participatory design activities and agile software development," in *Human Benefit through the Diffusion of Information Systems Design Science Research: IFIP WG 8.2/8.6 International Working Conference, Perth, Australia, March 30–April 1, 2010. Proceedings*. Springer, 2010, pp. 303–316.

[41] M. J. Muller, "Retrospective on a year of participatory design using the pictive technique," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1992, pp. 455–462.

[42] S. Davidoff, M. K. Lee, A. K. Dey, and J. Zimmerman, "Rapidly exploring application design through speed dating," in *UbiComp 2007: Ubiquitous Computing: 9th International Conference, UbiComp 2007, Innsbruck, Austria, September 16-19, 2007. Proceedings 9*. Springer, 2007, pp. 429–446.

[43] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[44] P. L. Gorski, Y. Acar, L. Lo Iacono, and S. Fahl, "Listen to developers! a participatory design study on security warnings for cryptographic apis," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–13.

[45] "Miro," https://miro.com/, Last accessed 2024-07-07.

[46] T. Zimmermann, "Card-sorting: From text to themes," in *Perspectives on data science for software engineering*. Elsevier, 2016, pp. 137–141.

[47] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[48] P. Pirolli and S. Card, “Information foraging in information access environments,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 1995, pp. 51–58.

[49] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, “An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 2, pp. 1–41, 2013.

[50] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath, “To fix or to learn? how production bias affects developers’ information foraging during debugging,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 11–20.

[51] P. Kather and J. Vahrenhold, “Is algorithm comprehension different from program comprehension?” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 455–466.

[52] M. X. Liu, J. Hsieh, N. Hahn, A. Zhou, E. Deng, S. Burley, C. Taylor, A. Kittur, and B. A. Myers, “Unakite: Scaffolding developers’ decision-making using the web,” in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, 2019, pp. 67–80.