An Exploratory Study of Sharing Strategic Programming Knowledge

Maryam Arab
George Mason University
Fairfax, VA, USA
marab@gmu.edu

Thomas D. LaToza
George Mason University
Fairfax, VA, USA
tlatoza@gmu.edu

Jenny Liang
University of Washington
Seattle, Washington, USA
jliang9@cs.washington.edu

Amy J. Ko
University of Washington
Seattle, Washington, USA
ajko@uw.edu

ABSTRACT

In many domains, strategic knowledge is documented and shared through checklists and handbooks. In software engineering, however, developers rarely share strategic knowledge for approaching programming problems, in contrast to other artifacts and despite its importance to productivity and success. To understand barriers to sharing, we simulated a programming strategy knowledge-sharing platform, asking experienced developers to articulate a programming strategy and others to use these strategies while providing feedback. Throughout, we asked strategy authors and users to reflect on the challenges they faced. Our analysis revealed that developers could share strategic knowledge. However, they struggled in choosing a level of detail and understanding the diversity of the potential audience. While authors required substantial feedback, users struggled to give it and authors to interpret it. Our results suggest that sharing strategic knowledge differs from sharing code and raises challenging questions about how knowledge-sharing platforms should support search and feedback.

CCS CONCEPTS

• **Human-centered computing → Systems and tools for interaction design**;

KEYWORDS

Programming strategies, Knowledge sharing

ACM Reference Format:
Maryam Arab, Thomas D. LaToza, Jenny Liang, and Amy J. Ko. 2022. An Exploratory Study of Sharing Strategic Programming Knowledge. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3491102.3502070

CHI '22, April 29-May 5, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-9157-3/22/04.
https://doi.org/10.1145/3491102.3502070

1 INTRODUCTION

Programming is hard. Programming languages can be difficult to learn [33]: APIs require immense domain knowledge to master [49], tools often pose steep learning curves [32], and developers regularly encounter novel problems which they must understand and solve creatively. And while new ways to address these challenges are constantly created (e.g., Stack Overflow answers to fill gaps in documentation and share reusable design patterns [29], YouTube videos that teach new coding technologies [36], online communities that offer mutual support [52]), *new* technologies result in an endless set of new problems for developers to solve.

Because of this problem-solving burden, one form of programming knowledge is essential to developer success and productivity: *programming strategies*. Prior work defines strategic knowledge in programming as any high-level plan for accomplishing a programming task, describing a series of steps or actions to accomplish a goal [24]. For example, consider the two strategies shown in Figure 1, which include step-by-step approaches to debugging the source of a wrong value or conducting a code review. In this paper, we define the term programming strategy broadly, encompassing problem solving strategies which may differ along a number dimensions: their length, completeness, level of detail, and generality. Strategies may be very specific to a particular programming tool or technology, explaining how to use it effectively, or offer very general approaches to solving problems across many tools and technologies. Strategies often require prior knowledge to use, and may differ in how much knowledge is already assumed and how much is explained within the strategy itself.

Research suggests that "experts seem to acquire a collection of strategies for performing programming tasks" like these over time and that strategies "determine success more than does the programmer's available knowledge" [17]. Recent studies have confirmed this, demonstrating that when developers are given explicit programming strategies that are known to be effective, the effectiveness of their work increases by making them more systematic and efficient [23, 24].

Many disciplines outside of software engineering have found ways of externalizing and sharing strategic knowledge to achieve such benefits. For instance, the Civil Engineering Handbook [9] describes numerous methods that frame civil engineering skills and provides examples of how to apply them to solve problems in the domain. Standard Operating Procedures (SOP) externalize and share

Strategy1: Fault Localization

1. Locate the lines that could have produced the wrong faulty value.
2. Check each line for errors or faulty values: check if the line is executed using a logging statement or a breakpoint.
3. If the line is defective and incorrectly compute the value, then you found the defect.
4. If the line itself was not defective, check the values it used to execute.
5. If there is a new faulty value corresponding to the current line, repeat steps 1-4 with the new faulty value until there are no more faulty values to look at.
6. If you haven't found your defective line, you might've made a mistake above! Check your work and start over.

Strategy2: Code Review

1. Look at each line of code contributed.
2. If the line is removed, verify why the line is removed.
3. If the line does not have a clear reason to be removed, ask why. Use the contributor's response to decide whether to keep the line.
4. Otherwise, if the line is added, check its style, documentation, logic.
5. If the added line has style or documentation issues or if there is an obvious bug, constructively comment on how the contributor can improve.
6. Next, verify that contribution is rigorously tested by using the software or looking at code coverage statistics.
7. If there is a defect or a lack of tests, constructively comment on how the coder can improve.
8. Finally, comment on what you think that the contributor did well on.

Figure 1: Examples of informal, explicit programming strategies for localizing a defect and performing a code review. Such strategies are rarely shared by developers online, even though they can be crucial to successful, productive programming.

knowledge, formalizing complex operations in military, health, and safety settings such as landing a plane or repairing a nuclear power plant [54]. SOPs offer step-by-step instructions, supporting users to be more efficient by preventing errors and regulating individual, team, and organizational behavior. Similarly, Gawande's *Checklist Manifesto* [15] argues that checklists of actions and states to verify help secure against medical errors, reduce complexity, and enhance performance in medical and surgical procedures. Through these various forms of strategic knowledge, users in many domains benefit from explicit sharing of methods to solve common problems, structuring their work by providing them reminders, guidance, and evidence-based methods for solving problems.

In software engineering, however, strategic programming knowledge remains largely inaccessible. Prior work suggests that it is largely gained through extensive experience or direct instruction [37]. And while modern software development is inherently social — with developers asking and answering questions on Stack Overflow [21, 30], sharing knowledge and expertise through social media, portals, and online chat [3, 8, 34, 44, 46], and attending social gatherings to exchange career and technical knowledge [8, 43, 44] — there is little evidence that developers share their strategic knowledge like the kind in Figure 1 in these settings. Rather, most platforms and communities focus on sharing code examples [21, 29, 50], knowledge on how to maintain, configure, or troubleshoot broader IT issues [1, 2], and tutorials that teach new technologies [36]. Such sharing has empowered developers to *construct* programs, but offers little guidance on how to orchestrate the problem solving that arises in the process of this construction, such as testing, debugging, program comprehension, and design.

One possible reason for the lack of sharing is that strategic knowledge might be *tacit*, in that it is situated, only effectively learned in context, and challenging to articulate explicitly to others [38]. If strategic knowledge in programming is tacit, developers may know how to solve various problems, but only when they enact that knowledge and not in a form that they can articulate and share. However, it is also possible that strategic knowledge is *not* tacit: prior work suggests that effective programming is a self-regulated, highly conscious activity [28, 39], suggesting it may be possible for experienced developers to articulate their problem-solving strategies. Moreover, some software development methodologies encode strategic knowledge explicitly. For example, practices such as test-driven development [6] provide relatively explicit problem-solving steps, demonstrating that, with effort, problem-solving processes might be able to be made explicit. Software developers who have such knowledge may just not think to share their knowledge, or know how or where to share it.

In this paper, we investigate why developers might struggle to share strategic knowledge. We hypothesize this knowledge is not tacit and can be articulated, but that there may be other barriers to writing and sharing strategies that prior work has not yet uncovered. Specifically, we examine:

- RQ1: What challenges do developers experience in explicitly articulating strategies?
- RQ2: When developers make use of explicit strategies written by others, what challenges do they face?

- RQ3: When experienced developers receive feedback from users of their strategies, what challenges do they face in improving their strategies?

To answer these questions, we simulated a programming strategy knowledge-sharing platform with 34 developers. Experienced developers were asked to articulate strategies, other developers used these strategies on programming tasks and provided feedback, and authors attempted to use the feedback and reflected on how they might revise their strategy in response. In the rest of this paper, we further ground our research questions in prior work, then describe our method, analysis, and results in detail. We end with a discussion of the implications for supporting the sharing of programming strategies.

We found that it *is* possible for experienced developers to share their programming strategies explicitly. Strategies varied greatly in detail, from high-level descriptions of processes captured in a few lines to elaborate procedures containing multiple sub-strategies focused on separate sub-goals. Authors experienced challenges generalizing their strategies to cover variation in strategy users' expertise, mirroring challenges by users. Strategy authors found the feedback they received from users helpful in improving their strategy, particularly in helping highlight expert blind spots. These results illustrate the potential for sharing strategic programming knowledge to harness the knowledge of experienced developers.

2 BACKGROUND

Social scientists studying the nature of knowledge sharing identify two forms of knowledge: *tacit* knowledge and *explicit* knowledge. Tacit knowledge is implicit, acquired from experience, and constitutes expertise. Explicit knowledge, in contrast, is easily transferred through written or natural language. While explicit knowledge can be easily articulated and communicated, tacit knowledge is not easily shared [53], and if it is, is costly and slow to externalize [18, 48]. Yet, at the same time, there is increasing evidence that tacit knowledge is an "important strategic resource that assists in accomplishing a task" [55]. Eliciting experts' tacit knowledge helps novices to increase knowledge and competence faster. Moreover, in organizational contexts, experts who leave an organization may result in knowledge loss that is costly and time-consuming, or impossible, to replace [26], which might be reduced by eliciting more tacit knowledge.

Studies of knowledge sharing in software engineering illustrate the widely varied difficulties which may occur in sharing software engineering knowledge. Design patterns are pervasive in code, but still require considerable effort to organize, describe, and disseminate (e.g., [14]). Architectural styles are common but implicit and take effort to identify, name, and describe [5, 11, 42]. As software teams generate knowledge, internal tools are often required to help developers externalize, organize, and share knowledge to facilitate collaboration and coordination [22]. In contrast to these more abstract embodiments of knowledge, sharing artifacts such as code and tutorials appears to be much easier [3, 35, 40, 46, 51]. The ubiquity of sharing on Stack Overflow [21, 30] and social media like Yammer, blogs, LinkedIn, and Twitter [8, 44, 46] reinforce general findings on knowledge sharing: when knowledge is explicit, it will be shared easily and widely.

Programming strategies are a central component of programming expertise [4, 17, 27]. Strategies can express a process to decide when to reuse code [47] as well as approaches to debugging (e.g. backwards or forwards reasoning, input manipulation, and intuition) [7]. Having an effective programming strategy can have more of an impact on task success than a programmer's knowledge of plans, design patterns, or other expertise [17, 24]. Using effective programming strategies may increase task success, reduce task time, and allow developers to work in a more systematic and structured manner [24]. Slicing strategies enable developers to better understand the problem [12] while also improving fault localization [10]. Teaching novice programmers a strategy to trace program execution improves comprehension [57]. Programming strategies can be represented in an explicit form and taught to novice developers [23, 37], offering an area of opportunity to explore knowledge sharing through programming strategies.

Closely related to programming strategies are *programming plans*. Soloway and Ehrlich's 1984 work defined programming plans as "program fragments that represent stereotypical action sequences in programming" [45]. Unlike programming strategies, which describe actions a *developer* will take, such as retrieving some information, making a decision, or configuring a tool, a *programming plan* captures abstract patterns of computation that may be found embodied in code (e.g., iterate over a collection). Much work has investigated programming plans, showing that they shape developers' programs independent of language [19] and that learning them is dependent on a robust understanding of a language's semantics [56]. Programming plans are specifically concerned with algorithm composition and design, and not with procedures for solving the many varied problems that arise in software engineering.

3 METHOD

In this paper, we investigate fundamental questions about the potential to share strategic programming knowledge. We examine the ability of experienced developers to write and express strategies explicitly, the challenges developers face in making use of these strategies, and the prospects for improving expressed strategies through feedback. To answer these questions, we conducted a study in which we simulated a knowledge-sharing platform. Experienced developers authored strategies, and less experienced developers used these strategies to complete programming tasks. Our study consisted of three phases. In the first phase, the first group, which we will call *authors*, each wrote a strategy for a task. In the second phase, the second group, which we will call *users*, each tested two of the authored strategies on two different tasks and provided feedback and comments. In the third phase, each author received the comments and feedback from the two users and was asked to elaborate on challenges in addressing the feedback. Our study was approved by the Institutional Review Boards of both of our universities. Our replication package, including all of the study materials as well as the anonymously collected data, is publicly available.[1]

[1] https://zenodo.org/record/5497775#.YToILtNKhTY

3.1 Tasks

In selecting tasks, we had several objectives. We needed the tasks to be familiar, so that experienced developers would be able to write strategies for them. Simultaneously, we needed the approaches for succeeding to be variable enough to observe a diversity of strategies. One approach to investigating strategy authoring would be to ask developers to write down a strategy of their choice. However, this might make it challenging to identify strategy users who would need this strategy and who had the appropriate expertise and limits the ability to compare alternative strategies for the same task. Therefore, we selected tasks for which we believed authors could write strategies and for which we could identify relevant users and contexts. We sought tasks which were neither too hard, given the limited time available to participants, or too easy, obviating a strategy's need.

We conducted nine pilot study sessions with five candidate tasks. In the pilots we asked them about the difficulties they faced about the task itself and the nature of authoring. We converged toward three tasks embodying common front-end web development activities: 1) *Chrome Profiler*: using the Chrome Profiler to improve a website's performance and identify the components responsible for slow performance; 2) *Error Handling*: verifying the robustness of error handling logic in a front-end web application; 3) *CSS Debugging*: debugging an arbitrary CSS problem on a web page with an incorrect visual style element. We also conducted three pilot study sessions to refine the testing phase.

Asking developers to use each strategy on authentic tasks of their own would have been ideal. However, finding developers at the moment they were encountering these problems in a real context proved infeasible. Therefore, we instead developed three programming tasks in which users could apply authored strategies. In the *Chrome Profiler task*, we downloaded a JavaScript application with performance issues involving moving images and buttons for adding, removing, and stopping moving images. Users used an explicit strategy to determine the cause of the performance issue and how to resolve it. In the *Error Handling task*, we developed a JavaScript application containing several errors that could occur. The users used an explicit strategy to identify potential errors that might occur and ensure that each of these error conditions was handled appropriately. In the *CSS Debugging task*, we developed a front-end web application with several visual style defects, including an incorrect header color, wrong border color, and incorrect background color for buttons. Users used an explicit strategy to find the causes of the defects.

3.2 Strategy Description Notation

There are many forms in which authors might share their strategic programming knowledge. We considered an unstructured natural language, a natural language with hierarchical bulleted lists (as shown in Figure 1), and other formats. We introduced a structured strategy writing language, Roboto [24], to help authors make their strategies as explicit as possible. Roboto is primarily a natural language, but includes simple control flow constructs such as conditionals and loops to help strategy users to be more systematic and comprehensive. We suggested it as a guideline for authors to organize their thoughts and communicate more precisely to strategy users. We did not require strict conformance to Roboto notation in this study, and participants were free to use the approach of their choice.

```
STRATEGY localizeWrongValue(wrongValue)
  SET 'lines' to all of the lines of the the program that could have produced 'wrongValue'
  # We'll check each line for errors, or for faulty values.
  FOR EACH 'line' IN 'lines'
    # Use a logging statement or a breakpoint to verify that this line actually executed.
    IF 'line' executed
      # Does the line incorrectly compute the value? If so, you found the defect!
      IF 'line' is defective
        RETURN 'line'
      # If the line itself wasn't defective, maybe one of the values it used to execute was
      # defective.
      SET 'badValue' TO any incorrect value used by the line to execute
      IF 'value' isn't nothing
        RETURN localizeWrongValue('badValue')
  # If you made it to this line, then you didn't find the cause of the wrong value. Is it
  # possible you made a mistake above? If so, check your work and start over.
  RETURN nothing
```

Figure 2: An explicit representation of a programming strategy guiding a developer's manual work to localize a defect

Figure 2 lists an example Roboto strategy, illustrating how to localize a defect by following data dependencies. Strategies consist of statements describing what the developer should do next. These include performing a specified action, gathering information, or making a decision about how to proceed. Statements in Roboto can be one of six forms: Action, Definition, Conditional, Loop, Return, or Call [24]. Additional details about each statement can be included through comments, indicated with a hash symbol before a statement. Strategies may also include preconditions for using the strategy, listed before the strategy declaration. Preconditions describe the knowledge or familiarity a user should have with technologies, resources, languages, tools, environments, and platforms.

3.3 Participants

To recruit participants to author strategies, we sought developers with at least three years of experience in front-end web development in any technology stack. We recruited alumni of our institutions' working as professional web developers. We required author participants to be familiar with at least one of the three tasks by self-evaluating their expertise with each task to decide if they were qualified to author a strategy. Twelve invited authors did not consider themselves qualified and withdrew from the study, reporting that they had insufficient familiarity with front-end web development, insufficient experience and confidence to write strategies, or insufficient time.

To recruit strategy users, we sought developers with a diverse range of programming experience to understand the range of possible difficulties developers with different skills might face in using a strategy. We required users to be at least 18 years old and be familiar with front-end web development technologies, including JavaScript, HTML, and CSS. We recruited users from the alumni of authors' institutions and graduate students in computer science and software engineering at both institutions.

We collected demographic data from both the authors and users about their prior industrial software and web development expertise, the number of software and web applications they had worked on, and the largest or most complex application they had developed. In addition, we asked them to describe their programming and work experience in a few sentences as well as include a link to any professional profile they might have (e.g., LinkedIn, GitHub.)

Participants included 19 *authors* (identified as A1-A19) and 15 *users* (identified as U1-U15). Authors ranged from 3 to 48 years of

experience in software or web application development (median 9 years). We asked participants to separately report the number of web-based applications and software applications they had developed. They reported having developed between 0 and 1,000 software applications (median 10) and 2 to 40 web applications (median 7). Authors worked in varying roles, including senior software developer and software architect positions, with diverse expertise in technologies such as full-stack web development, data visualization, educational games, network monitoring, and virtual reality. Users ranged from 6 months to 9 years of programming experience, with a median of 3 years.

3.4 Data

At the end of each phase of the study, we collected survey responses. After authoring a strategy in phase one, authors completed a survey about the difficulties they faced. To prompt authors to reflect on specific difficulties and help them recall their experiences, we brainstormed potential difficulties. Authors rated their level of agreement with seven potential difficulties (collected from the pilot study) on a 5-point scale and then briefly described their experiences with each. These included translating their thoughts into words, making the strategy understandable for novices, deciding when the strategy has covered all scenarios and edge cases, the effort and time required, concentrating on the task, and using Roboto and the authoring guidelines. The given prompts are shown in Table 1. Authors were then prompted to share any other difficulties they experienced.

After finishing using each of the two strategies in phase two, we asked strategy users to consider five questions about their experiences. We asked what made the strategy challenging to use in general or for a specific step, aspects of the strategy the user believed were missing, additional information or details which would make it easier to follow, aspects which made it confusing or ambiguous, and any additional challenges they faced.

In the third phase, we sent strategy users' comments and feedback to each strategy's author. Because each strategy was used with two users, each author received two sets of responses. For each comment in each user's response, we asked authors to describe the extent to which it was understandable, what might make it hard to address, what might have led them to have not initially addressed it when authoring the strategy and forgot to consider.

3.5 Procedure

The study consisted of three phases and was conducted entirely asynchronously and remotely, through email and dedicated web pages per phase. We selected this design to reflect the future context in which we expect strategy sharing to occur as well as to best accommodate the schedules of experienced developers. Figure 3 overviews each phase of the study and part of the study process we conducted for a CSS-Debugging task.

3.5.1 Phase One: Authoring Strategies. After agreeing to participate and selecting one of the three tasks, authors started the study by reading a tutorial about programming strategies, illustrating a strategy for lifting up state in React. They then completed a tutorial explaining the syntax of the strategy description language Roboto. To help them understand how to write strategies, the authors read several guidelines for authoring strategies. The guidelines suggested defining the strategy step by step; describing required tools, environments, and knowledge; using comments to elaborate; avoiding wasted work; including explicit restarts and rationale; and encouraging strategy user externalization.

Authors next received their selected task and wrote their strategy in a text editor panel. If authors had difficulties completing the task, they were encouraged to email the experimenters for clarification. Immediately to the right of the strategy editor panel, the authors could view a sample Roboto strategy. We believed an example would help authors recall the language syntax, if they chose to use it. To give the authors flexibility in how to express their strategy, we did not apply any syntax checking or highlighting in the text editor panel. After finishing their strategy, authors then completed a survey on the difficulties they faced and completed the demographic items. Authors had one week to complete phase one, with a series of reminders and extensions given on days 5, 7, and 12.

3.5.2 Phase Two: Using Strategies. After agreeing to participate, users read an introduction to programming strategies and completed the same Roboto tutorial as authors. Users then used two different strategies written by authors on programming tasks. For each task, users read a description of the task and one of the authored strategies. Users received a link to an online IDE configured with the code for the task as well as the task description. The users were then asked to complete the task using the strategy step by step. Users then completed several survey items about the challenges they faced in using the strategy. These included: what made it challenging, confusing, or ambiguous to work with the strategy; what is missing; and what additional information, details, or features would make it easier to follow. Finally, we asked users to describe any other challenges they faced. After completing the first task, users then used a second strategy to complete a second task and again completed the survey items on its challenges. Finally, users completed the demographic items. The users had four days to complete and submit phase two, with a series of reminders and extensions given on days 3, 4, and 9. If they did not complete the task and survey after day 10, we dropped them from the study.

After each user submitted their feedback, one of the experimenters read it. If it was unclear, the experimenter asked follow-up clarification questions through a shared document containing the author's strategy and their feedback. Multiple rounds of follow-up communication were conducted until the experimenters entirely understood the user responses. Users who completed the study received a $30 Amazon gift card.

3.5.3 Phase Three: Revising Strategies. In phase 3, users' responses on the challenges they faced with using a strategy were sent to the corresponding strategy author. Responses were sent using a shared document. Each user response was followed by three survey questions for the author: (1) Does this comment make sense to you; why or why not? (2) What, if anything, makes this comment hard to address? (3) Was there an aspect related to this comment of your strategy, which you forgot to consider; what made it hard to consider? Authors who completed the study received a $40 Amazon gift card.

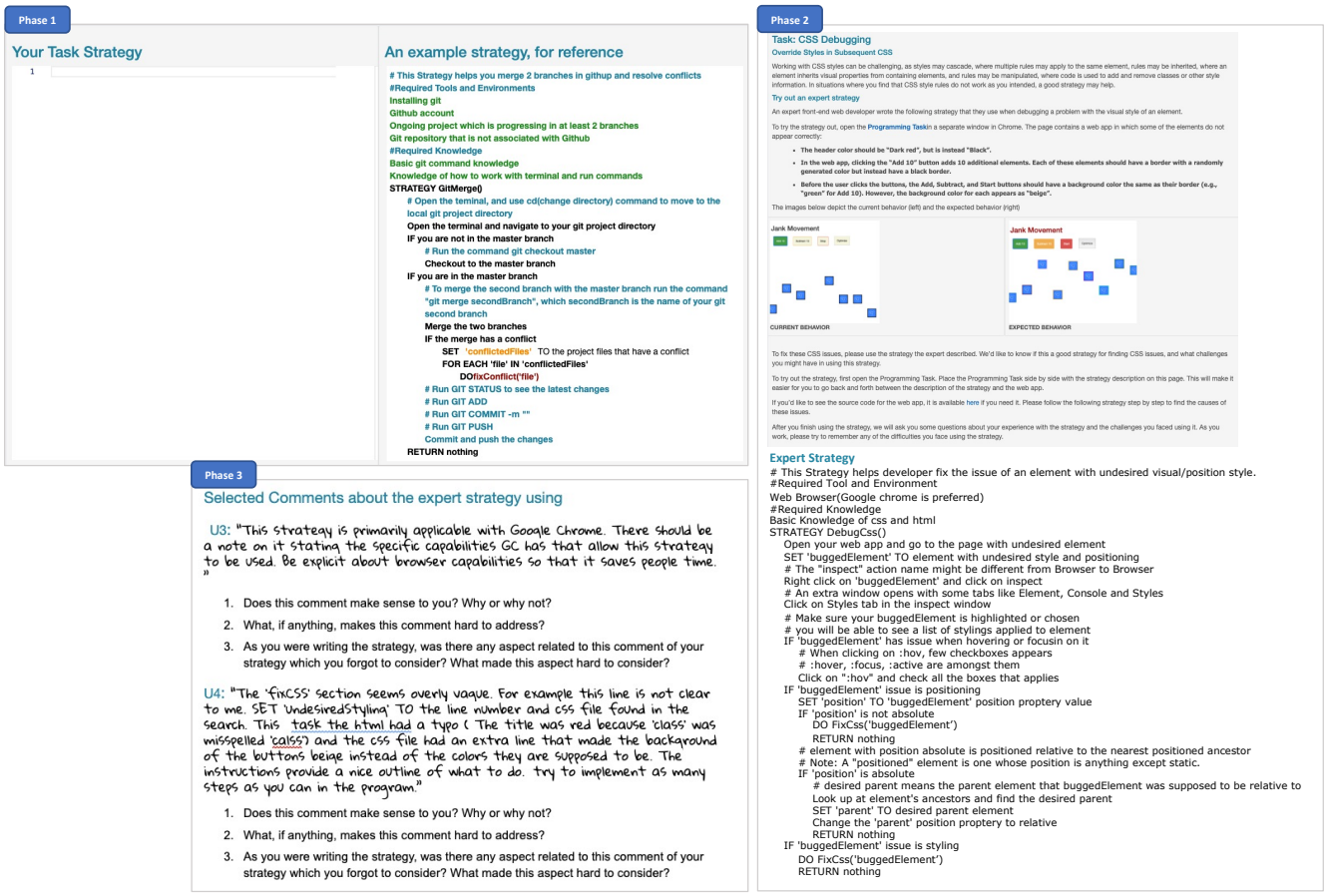

Figure 3: Participants in the three study phases were given web-based materials to author strategies (Phase 1), use strategies on a programming task (Phase 2), and comment on making strategy revisions (Phase 3).

3.6 Analysis

Our analysis focused on answering three research questions. First, as developers articulated their strategies, we examined the difficulties they faced (RQ1). Second, as other developers used these strategies on defined related programming tasks, we examined the challenges that strategy users experienced (RQ2). Third, based on the feedback strategy authors received from users, we examined the potential challenges authors experienced in using feedback to improve their strategies (RQ3).

In analyzing participants' descriptive responses to the survey prompts, we first extracted any difficulties they explained separately from the question topic. To reduce potential priming effects of the question prompts on the results, we ignored the prompts' topic in our analysis. Responses to Likert scale items in the first phase were analyzed separately.

To analyze the responses, we followed recent best practices in qualitative coding, which treats results as novel claims to be tested in future work, not as data to be quantified, and which recommends surfacing disagreements as an indicator of interpretation variance [20]. Following these guidelines, we first created a document containing all of the responses from the authoring and testing phases (excluding the prompts) for qualitative analysis [41]. In the first round of qualitative coding, three authors separately read each of the responses and inductively generated codes. The three paper authors separately identified difficulties, creating codes with a brief description. The three paper authors then individually labeled each response with zero or more codes. To aggregate these codes, the paper authors first compared the individually generated codes to identify codes with the same definition, adding them to the code book under a unique label. The three authors compared the codes, discussed instances of disagreement, and reached agreement by either adding or removing the code from the code book. Disagreements largely stemmed from variation in how to scope codes, and not in the meaning of author or user statements; disagreements were therefore resolved by agreeing upon scoping. During this process, all remaining codes were found to convey unique challenges participants experienced and were added to the code book. Using the final code book, the authors then coded the responses in a second round. The authors then applied pattern coding to the final codes [31], which groups codes into several broader categories. This process was conducted for all three phases of the study.

```
# This Strategy helps you identify potential errors in a
# UI implementation and how to approach implementing error-handling for
# Required Tools and Environments:
A web browser for running the UI
A breakpoint debugger for the UI code (likely a JavaScript debugger)
# Required Knowledge
An understanding of the intended behavior and possibilities of the UI
An understanding of the UI code such that intended code paths can be identified
STRATEGY IdentifyAndHandleErrors ()
# Identify the errors, keep track of them
SET `errors to handle` TO IdentifyErrors ()
# Add error handling for them
FOR EACH `error` IN `errors to handle`
   DO HandleError(`error`)
STRATEGY IdentifyErrors ()
# Open the UI that you are testing
Open the UI that is being tested
SET error_sequences TO []
# Attempt to execute every possible sequence of inputs.
# For complicated UIs, there may be an impossible number of combinations
# Test the most likely sequences of events and consider streamlining UI
if this is the case
SET 'possible inputs' TO all actions the user can take in the UI
   FOR EACH 'input' IN 'possible_inputs'
      # Run each input, make sure to test every combination
      DO `input`
      IF UIInErrorState ()
         Record input sequence that caused it, add to error sequences
   RETURN error_sequences
STRATEGY UIInErrorState ()
SET `is_error` TO false
IF the UI allows for inputs it should not
   SET `is_error` TO true
IF the UI is displaying incorrect information
   SET `is_error` TO true
IF the UI is frozen or unresponsive
   SET `is_ error` TO true
RETURN `is_error`
STRATEGY HandleError(`error`)
IF error can be prevented
   Update code to avoid error state
IF error cannot be prevented
   IF error is confusing to user
      Update UI to explain the error to user
   IF error locks up the UI
      Update component to reset UI or refresh page
```

Figure 4: An *Error Handling* strategy authored by A11 and used by U10 and U11.

4 RESULTS

Overall, we found that all 19 strategy authors were able to author an explicit strategy for the task they selected. Strategies varied in length from 4 lines to 78 lines, with a median length of 34 lines.

Strategies consisted of five main elements: specifying the knowledge and tools necessary to use the strategy, enumerating potential issues to investigate, determining if the issue applies to the situation at hand, offering a solution plan for addressing the issue, and applying it to edit the code. Strategies varied in how many of these they included. For example, Figure 4 lists a strategy written by A11 explaining how to handle errors. The first few lines (gray text) specify the required tools, environment, and knowledge that a user needs to use the strategy. The strategy is organized into multiple sub-strategies (each beginning with the strategy keyword in blue text) describing how to accomplish specific sub-goals. At the beginning of the first, "IdentifyAndHandleErrors," the user is asked to record potential errors in a collection variable, after first following the "IdentifyErrors" strategy to enumerate errors. Working through each potential error, the user is then asked to apply "HandleError". The user then determines if each of a set of specific issues may apply, with specific actions to take to resolve each issue.

In the following sections, we report challenges in sharing strategic programming knowledge, focusing on the challenges experienced by authors in making strategic programming knowledge explicit, the challenges experienced by users in using this knowledge to complete programming tasks, and the challenges experienced by authors in using feedback from users to improve their strategies.

4.1 RQ1: Challenges Authoring Strategies

To understand the difficulties authors faced in explicitly expressing their strategic knowledge, we analyzed the free responses given by authors in phase one to prompts to reflect on the authoring challenges they faced. We received 144 free responses, which included responses from each of the 19 authors to the 7 prompts as well as 11 responses to the other difficulties prompt.

The ordinal-scale agreement responses to the prompts are shown in Table 1. Most authors agreed that the strategy writing guidelines were helpful and that writing strategies took substantial concentration, effort, and energy. Responses to the remaining prompts were more varied, reflecting differences between authors in what they found to be challenging.

Through an analysis of authors' free responses, we identified 22 challenges in authoring explicit strategies, organized into five categories: finding the right scope, approaching writing a strategy, using the Roboto strategy language, the effort required, and taking the user's perspective. We discuss each in the following subsections.

4.1.1 Finding the right scope. Strategy authors reported six challenges in finding the right scope for their strategy (Table 2). Some found it difficult to be neither too general nor too specific. Others reported difficulties imagining the range of scenarios it should cover and not forgetting steps that might have become habitual and tacit for them. Other challenges included writing a strategy to address the large and complex problems that might be encountered and how to test that a strategy works in all cases. Authors also expressed challenges with writing the strategy to be flexible and appropriately respond to new information, which users might uncover while executing the strategy. We report two particularly interesting difficulties below.

Generalization. Authors mainly reported concerns that strategies could become too general to be helpful or too specific to be relevant to the the wide range of tasks and situations that might occur. These challenges were reflected in the widely varying level of detail authors chose to include in their strategies. For instance, Figure 5 lists a strategy where A8 included a number of details and handled several edge cases. Based on their experience, they reported:

> *"Some edge cases is hard to include in a strategy as a general recipe. Also, sometimes you need to show a demo or some sort of an example to make your point."* (A8)

Others reported that choosing a specific domain could reduce these difficulties. As many tools, languages, frameworks, and technologies vary across contexts, specifying a narrower context might simplify the difficulty in generalizing across contexts.

Testing. Few authors reported challenges testing that their strategy worked well in all possible cases:

> *"It's hard to know how one would safely conclude they've tested for all possible errors."* (A11)

Authors suggested that including a program to test their strategy might help identify missing steps, conditions, and details.

4.1.2 Effort required. Most of the authors reported that strategies were hard to write because of the time and effort required (Table 2). Some felt that creating a written strategy was more time consuming than verbal communication, while others found the required intense focus and concentration challenging. Authors varied in their response to this challenge, with one reporting that this work was inherently boring while many others reported finding it exciting.

Prompts	Agree	Neutral	Disagree
Strategy writing guidelines helps to effectively express strategies	79	21	0
Took a lot of concentration, effort and energy	74	21	5
Articulating is time consuming and boring	47	16	37
It's hard to write strategies in a way which are understandable for novice developers	47	26	26
Hard to translate thoughts & strategies to words	47	11	42
The Roboto language supports your ability to effectively express strategies	42	32	26
Terminating the strategy is hard. Recognizing what would be the last statement is hard	26	21	53

Table 1: The percentage of strategy authors agreeing or disagreeing with prompts. "Agree" and "Disagree" include Strongly Agree and Strongly Disagree.

```
# This Strategy helps developer fix the issue of an element with undesired visual/position style.
#Required Tool and Environment
Web Browser(Google chrome is preferred)
#Required Knowledge
Basic Knowledge of css and html
STRATEGY DebugCss()
Open your web app and go to the page with undesired element
SET 'buggedElement' TO element with undesired style and positioning
# The "inspect" action name might be different from Browser to Browser
Right click on 'buggedElement' and click on inspect
# An extra window opens with some tabs like Element, Console and Styles
Click on Styles tab in the inspect window
# Make sure your buggedElement is highlighted or chosen
# you will be able to see a list of stylings applied to element
IF 'buggedElement' has issue when hovering or focusing on it
    # When clicking on:hov, few checkboxes appears
    # :hover, :focus, :active  are amongst them
    Click on ":hov" and check all the boxes that applies
IF 'buggedElement' issue is positioning
    SET 'position' TO 'buggedElement' position property value
    IF 'position' is not absolute
        DO FixCss('buggedElement')
        RETURN nothing
    # element with position absolute is positioned relative to the nearest
       positioned ancestor
    # A "positioned" element is one whose position is anything except static.
    IF 'position' is absolute
      # desired parent means the parent element that buggedElement was supposed to be relative to
        Look up at element's ancestors and find the desired parent
        SET 'parent' TO desired parent element
        Change the 'parent' position property to relative
        RETURN nothing
IF 'buggedElement' issue is styling
    DO FixCss('buggedElement')
    RETURN nothing



STRATEGY FixCss(buggedElement)
# You can use filter input to search for it
# Or you can scroll through the styles manually
Search through the stylings to find where it gets its undesired value
SET 'undesiredStyling' TO the line number and css file found in the search
IF 'undesiredStyling' is not found
    # You will find all stylings applied to the element here
    # Once you found the stylings you were looking for
    # You can click small arrow to jump to the place it gets its value
    Click on Computed tab and use filter to search
    SET 'undesiredStyling' TO line number found here
SET 'perfectStyleList' TO an empty list of css properties
UNTIL buggedElement has desired styling
    # you can add or change different css styles to the element
    # it then applies instantly to element stylings
    Use element.Style to apply css to buggedElement
    add the style proptery to 'perfectStyleList'
    DO ApplyCssToElement(buggedElement, 'perfectStyleList')
```

Figure 5: Two of the sub-strategies included in the *CSS Debugging* strategy written by A8.

Authors felt that strategy authoring, similar to programming skills, requires time and effort to learn. These results are consistent with authors' responses in Table 1.

4.1.3 Perspective taking. Some authors found it hard to select an expected level of knowledge for the strategy user while ignoring their own knowledge level:

> *"It's difficult to know if somebody else would understand the instructions."* (A9)

A few authors felt that they needed to guess what questions users might ask first in approaching the problem and address this in their strategy. The ordinal agreement results show that about half of the authors found it challenging to write strategy in an understandable way for the novice developers.

4.1.4 How to approach writing a strategy. Authors expressed challenges with the cognitive process of explicitly articulating their knowledge, demonstrating the strategy to users without the use of external resources or aids, determining how to effectively frame solving the problem, and explaining choices between alternative approaches (Table 2). Two were particularly revealing.

Level of detail. Some authors found it hard to find and express the strategy with the right amount of detail for the level of user expertise. This mirrors substantial differences between strategies in the level of detail they included. Some posed the challenge as balancing brevity and detail:

> *"To make the description easy to follow and understand, I'd probably be leaving out a lot of edge cases and essential information."* (A11)

Others wished for examples to guide them in understanding the appropriate level of detail needed for different users:

> *"More examples targeted at various expertise levels would help."* (A9)

Externalizing strategic knowledge. Few authors found it hard to translate their thoughts into words. Some expressed challenges recalling past strategies they had used:

> *"I just need to remember all the situations I was in and how I resolved the issues."* (A2)

Some felt that describing strategies verbally for a specific audience would be much easier than writing them down:

> *"It is hard because, for many developers, we do not spend time to write; instead, we are focusing more on coding. If we are in a meeting and explain the way we do things will much easier than write them out in a document."* (A10)

4.1.5 Using a strategy description notation. While authors were not required to follow the strategy description notation's syntax, most tried to use it, and some found this difficult (Table 2). One reported that, just as with learning a new programming language, its novelty made it hard to learn within a limited time. Many reported that the incompleteness of the language constructs made it hard to express some aspects clearly and concisely. Others felt it would be simpler to express programming strategies using natural language. Two were unsure what value a structured notation offered, when it still needed to rely on natural language comments to explain steps. Others reported missing features in the editor, including a desire for it to include syntax highlighting. Among the challenges reported,

Difficulty	Description
Finding the right scope	
Generalization	Concerns about strategies being too general to be helpful or too specific to be relevant to many cases
Abstraction	Challenges imagining the range of scenarios to cover in a strategy
Completeness	Being consistent, structured, and planned and not forgetting steps that are habitual and tacit
Scalability	Scaling to large and more complex problems, which may make strategies hard to understand and use
Testing	Ensuring the strategy works well in all cases
New Information	Responding to new information during the task
Effort required	
Time	More time consuming than verbal communication
Concentration	Cognitively demanding, which requires freedom from distraction and can be frustrating and exhausting
Perspective taking	
Knowledge Level	Selecting the target knowledge level of the strategy user
Perspective	Adopting the perspective of the strategy user, with the necessary level of detail required
Approach writing a strategy	
Level of Detail	Finding and expressing the right level of detail to effectively explain the strategy
Process	Determining how to effectively frame solving the problem
Demonstration	Illustrating the strategy without demonstrating it on a real task or support for communicating necessary concepts
Tool Use	Communicating terminology and concepts necessary to use referenced tools
Externalization	Recalling strategies used in the past, externalizing and translating thoughts into words
Organization	Learning how to correctly structure the strategy with insufficient strategy examples
Usability	Ensuring that the authored strategy works well with real programs
Choice & Repetition	Explaining choices between alternative approaches and generalizing similar steps to reduce repetition
Strategy description notation	
Expressiveness	Expressing strategy in a way that is clear and concise with inadequate language constructs to do so
Formal Notation	Expressing ideas that are simple to say in natural language more formally in strategy description language
Novel language	Learning and using the novel strategy description language
Authoring Tools	Using language with missing support in strategy editor for syntax highlighting, code formatting, line breaks, toolkit

Table 2: Reported challenges explicitly articulating strategic programming knowledge, sorted from most to least frequently reported.

some authors included unsolicited positive feedback about their experiences writing programming strategies:

> *"It helps to communicate problems more clearly. Shows experience, makes us think out of [the] box."* (A4)
> *"It helped me think [about] what I should do for my work project."*(A10)
> *"Roboto gives it a standardized and structured format which could be easier for any developer to follow. "* (A1)
> *"Roboto language can make it more opinionated to write strategies. It is some kind of standardizing for writing strategies."* (A8)

4.2 RQ2: Challenges Using Strategies

To characterize the challenges developers face in *using* programming strategies written by other developers, we analyzed the 150 comments we received from strategy users. Through an analysis of user' feedback, we identified 11 challenges in using explicit strategies, organized into two broad categories: challenges related to the strategy and challenges related to the user's knowledge.

4.2.1 Strategy-related usage difficulties. Users reported eight challenges related to understanding and using strategies (Table 3). Some found it hard to understand what the strategy asked them to do and the relationship of this to their overall task. Most users needed more precise descriptions of how to take specific steps in the strategy. Some reported that they needed to read the strategy multiple times to understand what it asked them to do. Others found it challenging to understand if the strategy would work for the programming task at hand, as the strategy seemed to miss relevant edge cases. For example, Figure 6 displays challenges U3 expressed in their feedback (violet font) in understanding *unfamiliar terminology* (list-boxes, alpha, special characters), several challenges with *imprecise steps* lacking detail on how to perform the step, and *ambiguity* about why a step is required or where they need to perform the action.

Several of the challenges users experienced were directly related to those reported by the strategy authors:

- `Ambiguity.` Most users expressed confusion understanding the rationale behind performing actions in the strategy. They reported being confused about what each step tried to accomplish and why it was necessary. This reflects differences between authors in the amount of rationale and detail they chose to include. Other users were confused about the terminology that the authors used in the strategy. In other cases, there were errors in the strategy. In one case, a user was confused by a strategy which invoked a sub-strategy that was never defined.
- `Imprecise Steps.` Almost most of the users reported needing more detail to understand how to perform a strategy step, such as what code to refer to or how to take an action. This mirrors differences in the level of detail authors chose to include and the difficulties authors reported in choosing an appropriate level of detail (Section 4.1.4).
- `Tool Use.` Some users found it hard to use a required tool to perform the described action. They reported problems finding the features the strategy instructed them to use in the tools. This mirrors difficulties authors experienced describing the tool use actions that they wished users to take (Section 4.1.4.)

4.2.2 Knowledge-related usage difficulties. Several challenges reflected a mismatch between the level of knowledge assumed by the strategy and possessed by the user (Table 3). Some users felt that they had more knowledge and experience than the author and suggested strategies or steps they felt better accomplished the task. Others lacked sufficient knowledge and felt that the strategy lacked detail. This mirrors differences in the level of detail authors included as well as the challenges authors reported in finding the right level (Section 4.1.4) and adopting the user's perspective (Section 4.1.3).

4.3 RQ3: Challenges Improving Strategies

After returning users' feedback to the strategy authors, authors were asked to reflect on how feasible the issues would be to fix and what they believed had caused each of the challenges. An analysis of the reflections yielded seven challenges in revising strategies. Beyond the challenges reported, authors also reported that some of the feedback was constructive, comprehensive, and helpful. Authors sometimes agreed with the limitations users reported, such as suggestions on how to make their strategy more comprehensive.

Authors sometimes felt that the goal and scope of the strategy they were asked to write were not well-defined, leading to a mismatch between what they wrote the strategy to do and what the user expected. They suggested that offering an example or an image of the step could offer clarification by helping users find a mentioned section in the strategy. For example, Figure 7 lists A3's strategy for profiling in Chrome, for which U2 gave feedback:

> *"I used chrome but still I was not able to find the NET section to find the CSS component. It took me a long time to find the component."*

A3 agreed, explaining that they *" did not consider all the different sections the user would be looking for. There are a lot of sections in the profile, so it could be challenging and time-consuming to consider every deviation."* They believed that this would not be difficult to address in revising their strategy.

In the rest of this subsection, we discuss the seven strategy revision difficulties.

4.3.1 Incorrect use. Some authors disagreed with user feedback, viewing it as reflecting a mistake or misinterpretation in following their strategy. For example, A14 received feedback:

> *"The main challenge that I observed was knowing which css properties to look for, as I usually use libraries to style my work, instead of using custom css. The other challenge was that sometimes a property with a strike through could not be overwritten."* (U13)

In response, they reported:

> *"The first part of this comment is not valid because the [strategy] tester missed reading the comment for all css files. The second part also might have been misunderstood by [the] tester."* (A14)

4.3.2 Generalizability. Some authors realized that the context for which they wrote the strategy differed from the user's context when, for example, a tool did not support necessary steps. Some authors also understood that the goal and scope of the strategy they wrote were not well defined, leading to a mismatch between what they wrote the strategy to do and what the user expected. For example, U10, who used the error handling strategy in Figure 4, stated:

> *"The point of view of the strategy writer as a tester [is missing]. It was also too general as I said like a pseudo code for me."*(U10)

A11 responded that:

> *"It was hard to consider how to make my instructions specific when the scenario was so abstract. Mostly [the feedback makes sense], though I don't know how pseudo code would help the problem of the code being too general."* (A11)

Difficulty	Description
Strategy-related	
Missing Steps	Missing instructions or steps to solve the problem.
Ambiguity	Difficulty understanding why a step is necessary to reach the goal (missing the rationale behind a step)
Imprecision	Need for more detail in describing specific steps or what is required to use the strategy
Generality	Inapplicability to specific contexts, situations, or edge cases
Unfamiliar Terminology	Unfamiliar words without definition or description
Tool Use	Difficulty determining how to use a tool to perform a described action, including located referenced features in the tool
Environment	Difficulty reading and using strategy for the lack of environment features and strategy syntax highlighting
Rereading	Need to read strategy multiple times to understand it
Knowledge-related	
Inefficient	More effective ways to accomplish the strategy goal than that described
Unfamiliar concepts	Lack of familiarity with concepts used by the strategy
Inapplicable approach	Using a strategy that does not address the problem

Table 3: Reported challenges using programming strategies, from most to least frequently reported.

```
STRATEGY UserInput()
# Goal is to limit free-hand input and use strong typing
Use list-boxes as much as possible  What's a list-box?
Use Try-Catch and log errors
IF list-boxes cannot be used, validate textboxes
   Validate datatypes for all entries
   IF numeric reject alpha  It seems like it would be ok here to specify "alphabetic characters". Alpha can also mean a specific
                                                                                                    symbol if I'm being nitpicky.
   IF date, validate dates and reject invalid dates
   IF character, check for injection (SQL and script) AND  How? Probably by parametrizing queries/stored procedures?
      Check for special characters  Like What?
   FOR EACH 'textbox' On Page
       Validate
   IF Error Found, RETURN Error  To user? Where?
   ELSE, Submit Page
STRATEGY Database()
# Handle null values, data truncation, invalid datatypes
In Queries check values before performing Substring functions  Check what about the value? Too vague to be helpful.
Use stored procedures to limit SQL injection  Don't just limit SQLi. Prevent it entirely using prepared statements
                                              or parameterized queries, which actually can prevent it. Stored procedures
                                              do not prevent SQLi by default

Use IsNull (or NVL) for nulls.  Why? In what situation? Does this mean put a string "IsNull" in values that will be null in the
                                database instead of leaving them empty?
Check for blank values using len(trim(<columnName>)) = 0
DO NOT USE "where 1 = 1" in queries as this allows for SQL injection  Factually incorrect, see below.
Use Try-Catch and log all errors  Where? To console? To an internal error log? To somewhere only
```

Figure 6: An *Error Handling* strategy written by A7 and used by U3 and U4. The feedback from U3 is displayed in violet.

It was hard for authors to describe what to do in every possible situation when there were many possible situations. Some authors also felt that they could not provide additional detail without reducing the generality of their strategy. Author A5 stated:

> *"I felt that the original assignment asked me to be generic as possible. I could not be more specific without knowing the language (or at least a family of languages) used, and especially without knowing the architecture. This is a problem that also happens in the real world. Gaining context about the exact problem being solved is key when mentoring a less experienced programmer."* (A5)

4.3.3 Mismatched level of knowledge. Authors sometimes realized that they had misjudged the user's prior knowledge and faced difficulties writing strategies without knowing it. Assuming background knowledge made some steps easier to understand or unnecessary

```
STRATEGY ProfileComponent()
#Open your chosen web appliation in the Chrome Browser
Open the Chrome Browser
Navigate to your Web Application
Right Click and select 'Inspect'
Click on the Performace Tab
Click the Record Button as indicated
Perform Task on the web application utilizing the component
Click Stop
#A peak is a section of the flame chart where CPU is high
IF there are peaks in the flame chart
    FOR EACH peak
        Click on the peak
        Drag your mouse to highlight the entire peak
        Hover over the NET section to view the CSS component utilized
        Click on the Event Log in the bottom
        Select the longest Task
        Expand the Task
        Click on the Function Call to identify the component
        Look at the current frame state, network requests, animations
    in order to come to a conclusion of why the component is slow
RETURN NOTHING
```

Figure 7: A *Chrome Profiler* strategy written by A3 and used by U1 and U2

to explain. For example, the strategy in Figure 6 leaves most statements as fairly vague, leading the user to offer feedback on several statements (violet font). The author reflected:

> *"It tells me that the reader is not a web developer and not familiar with web control objects like list-boxes (drop-down list boxes) and the term alpha for alphabetical."* (A7)

Other authors stated that users should gain background knowledge through their investigation rather than through the strategy.

4.3.4 Unneeded or unrealizable situations. Some authors reported that some users' comments asked for a strategy to address situations or contexts that cannot occur and were not necessary. For example, U3 who used the strategy in Figure 6 believed there are some steps missing in the strategy, *"More detail on what to do, ex. "check" what? Security depth of knowledge is also missing, especially around SQL injection."* Author A7, however, reported that it is not necessary: *"What to do next is up to the developer as each situation is different."*

4.3.5 Hard-to-address feedback. Some authors viewed some of the users' feedback to be hard to address. Some reported that some aspects of the strategy were hard to explain or would take too much time. Other requests were impossible for authors to immediately satisfy in the strategy editor, such as requests for syntax highlighting or adding links to images. For instance, Strategy user U5 said:

> *"I believe color-coding would be extremely crucial as it would be visibly easier to see for the user in terms of what is code, preconditions, statements, and actions."* (U7)

4.3.6 Resistance to increasing detail. Users sometimes requested further detail about specific steps. Authors sometimes felt that adding this detail would make their strategy too long and thus harder for users to follow, signaling a tension between detail and assumed level of expertise.

> *"It is hard to write [a] set of instructions so thoroughly to address all types of scenarios in simple words so everyone can understand, especially in the first try."* (A8)

4.3.7 Ambiguous feedback. Authors sometimes felt comments left ambiguous exactly what the user requested. Authors often viewed broad requests for additional detail to be excessively vague. For example, one author wanted a more concrete example of the types of detailed information requested or the exact line number in the strategy where the user had gotten stuck.

> *"Not being specific and providing enough details in the comment about why the strategy action was ambiguous and didn't make much sense to them makes it hard to address."* (A1)

In other cases, users used a term or reference that the author did not understand. Some authors had difficulty understanding how the user was interpreting specific statements they wrote. Some proposed including a screenshot of users' work to help them understand what they were doing and where they were getting stuck.

5 LIMITATIONS

External validity. Our study differed from a work context in several ways. In using programming strategies to problem solve, participants did not use other resources they might normally use, such as asking a teammate for help [25]. Participants may face different types of challenges in their programming tasks. Authoring strategies in a known context is easier and less challenging. Finally, in simulating the characteristics of a hypothetical platform for sharing strategies, the ways users and authors interacted in the study may differ from a real-world platform. Users might experience difficulties that they did not experience in our study when working with longer or more challenging programming tasks. Platforms might incorporate different feedback mechanisms, such as multiple rounds of interactions between authors and users or different ways of incorporating feedback. Our results are thus limited in partially reflecting characteristics of the specific platform we simulated.

Construct validity. There are no widely accepted measures of prior knowledge in programming. This may have caused variations in authoring expertise, causing participants to write strategies for tasks where they had insufficient expertise. The ordinal survey questions proposed possible difficulties, which may potentially bias the authors to focus on the proposed challenges and forget to report other categories of challenges. Some of the challenges were reported only by one participant and may not be broadly applicable.

Authors were asked to use Roboto to help better structure their strategies, and as the results showed, it helped authors meet this goal. Using Roboto may be particularly beneficial for complex strategies with multiple edge cases and scenarios. However, the authors did experience challenges using the notation, which might be addressed through better tool support, tutorial materials, or language improvements. The Roboto syntax did not contain all constructs participants wished to use (e.g., else statement). Some statements occupied multiple lines, making them difficult to separate. Syntax denoting the end of a line might help address this issue. More broadly, Roboto inherently encourages a procedural approach to describing strategies step by step. Other alternative representations of a strategy, such as a more declarative, event, or rule-oriented approach, might lead to different ways to express strategies, which might vary in some of the challenges authors or users experience. However, in our study, we saw little evidence for authors themselves preferring an alternative notation, with most using an imperative style to describe strategies step by step with Roboto or using completely unstructured natural language.

Internal validity. We did not directly observe users as they examined how closely or carefully they followed the strategies. Strategy users might have abandoned a strategy, using alternate

strategies, potentially limiting the validity of their feedback. Additionally, authors may have forgotten or misremembered some of the challenges they faced after writing the strategy. Similarly, when a user got stuck in a step in the strategy, they might have discontinued using it, limiting the difficulties they reported. In our design, the authors wrote a strategy for a specific type of task, which users then performed. We did not examine difficulties in identifying or choosing relevant strategies for a specific task. Some participants were unfamiliar with the concept of a strategy, and we, therefore, provided training materials. While more experienced developers might not need this training, inexperienced developers might have benefited from further training, which might have reduced the difficulties they experienced.

6 DISCUSSION

This paper examined the challenges of sharing strategic programming knowledge. We found that it *is* possible for experienced developers to share their programming strategies explicitly, consistent with the prior work that suggests that programming is a self-regulated and highly conscious activity [28, 39]. We found that strategic knowledge is, to some extent, tacit, as the authors often found strategies difficult to express with sufficient detail. It is also challenging to be used generality by users with varying expertise and needs. However, we found that, with sufficient effort, authors can succeed. Many factors may impact this difficulty, including authors' pedagogical skill in teaching, the complexity of the task, the frequency or recentness in which authors have used the strategy. More work is necessary to better understand these factors and their ultimate impact, particularly across authors with varying levels of expertise.

The authors' strategies varied greatly in detail, from high-level descriptions of processes captured in a few lines to elaborate procedures containing multiple sub-strategies focused on separate subgoals. Authors experienced challenges generalizing their strategies to cover variation in strategy users' expertise, mirroring challenges by users, including strategy ambiguity, imprecision, scope, and clarity. Many reported needing to re-read strategies multiple times to comprehend them. Authors were often surprised by strategy users' feedback, not anticipating gaps in knowledge, misinterpretations, and desire for additional detail. Strategy authors found the feedback they received from users helpful in improving their strategy, particularly in helping highlight expert blind spots. These results illustrate the potential for sharing strategic programming knowledge to harness the knowledge of experienced developers.

Alternative mechanisms for eliciting strategies might help to address some of the challenges that authors experience. For example, authors might instead be asked to write several concrete strategies for specific tasks and, only after doing so, be asked to generalize them into a single, more general strategy. Alternatively, strategy writing might be crowdsourced, where similar strategies might first be written by different authors and then combined and generalized. Exploring more effective prompts and workflows for eliciting strategies is essential for future work.

One interpretation of these findings is that strategy authoring is less like sharing code and more like instructional design in teaching [13]: it appears to require a strong awareness of users' prior knowledge, knowledge of variation in that prior knowledge, and careful attention to scaffolding skill development. From this perspective, it becomes clearer why the authors in our study faced the difficulties they did: they did not know whom they were teaching and what knowledge they had. Perhaps more importantly, they likely had no instructional design expertise. This interpretation of the results would suggest that the ideal skill set for authoring programming strategies would be those who both know the strategies well and have the instructional design expertise to carefully craft various strategies that serve audiences with different levels of prior knowledge. In educational research, such expertise is called *pedagogical content knowledge* (PCK), which simply refers to the knowledge required to teach a particular knowledge [16]. This might suggest that there is PCK for successfully authoring programming strategies, much like there is PCK for teaching math, science, writing, and other subjects. Future work might consider studying developers who have experience supervising teams, where they might have been likely to develop programming strategies PCK while mentoring and guiding more junior developers.

Another interpretation of our findings is that, while programming strategies may not be tacit, developers do not represent them in a formal, rigid, or structured fashion that is *easy* to translate into explicit forms. Authors in our study expressed difficulty in concentration and writing, suggesting that while developers are aware of strategic knowledge to the degree that makes it shareable, there is an effort to find words, ideas, concepts, and structures that faithfully capture strategic knowledge in a form that others can use. In contrast to the instructional design interpretation, this translation interpretation suggests that knowledge cannot simply be "exported" to text but must be recalled, organized, articulated, and revised. This suggests that, unlike sharing code, which may require less generalization and synthesis, sharing strategies may be a highly effortful cognitive process requiring motivation, practice, and focus.

If either or both of these interpretations are true, there are several implications for designing mechanisms and platforms for sharing programming strategies. Future work might explore tools that help authors brainstorm, structure, evaluate strategies, and better leverage multiple forms of media to clarify strategic procedures. Future work could continue refining notations like Roboto [24], which allow for some degree of informality and flexibility in specifying strategies. Future platforms for sharing strategies may need sophisticated support for soliciting user feedback, offering strategies with multiple levels of detail, and helping address gaps in users' prior knowledge for a strategy, perhaps by linking to other strategies or other resources. Authors may also vary in their pedagogical styles, suggesting the need for a diversity of strategies for the same programming problems, allowing users to find strategy authors whose voice and teaching resonate. Moreover, all of this diversity suggests the need for novel forms of strategy search, helping connect users with strategies that match the programming problem they are facing and their prior knowledge.

If future work can address these challenges, there is substantial potential for developers to share their strategic programming knowledge, enabling hard-won knowledge and expertise created through years of experience to be broadly shared for others' benefit.

REFERENCES

[1] Mark S. Ackerman and Thomas W. Malone. 1990. Answer Garden: A tool for growing organizational memory. *ACM SIGOIS Bulletin* 11, 2-3 (1990), 31–39. https://doi.org/10.1145/91474.91485

[2] Mark S. Ackerman and David W. McDonald. 1996. Answer Garden 2: Merging organizational memory with collaborative help. In *ACM Conference on Computer-Supported Cooperative Work*. 97–105.

[3] Nicolette Bakhuisen. 2012. Knowledge sharing using social media in the workplace. *Unpublished Master thesis. University Amsterdam, Amsterdam* (2012).

[4] Sebastian Baltes and Stephan Diehl. 2018. Towards a theory of software development expertise. In *ACM Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 187–200. https://doi.org/10.1145/3236024.3236061

[5] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software architecture in practice*. Addison-Wesley Professional.

[6] Kent Beck. 2003. *Test-driven development: By example*. Addison-Wesley Professional.

[7] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? An experiment with practitioners. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 117–128. https://doi.org/10.1145/3106237.3106255

[8] Thomas Chau and Frank Maurer. 2004. *Knowledge sharing in Agile software teams*. Springer Berlin Heidelberg. 173–183 pages.

[9] Wai-Fah Chen and JY Richard Liew. 2002. *The civil engineering handbook*. Crc Press.

[10] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. 1996. Critical slicing for software fault localization. In *ACM International Symposium on Software Testing and Analysis*. 121–134. https://doi.org/10.1145/226295.226310

[11] Davide Falessi, Giovanni Cantone, Rick Kazman, and Philippe Kruchten. 2011. Decision-making techniques for software architecture design: A comparative survey. *Comput. Surveys* (2011), 33:1–33:28. https://doi.org/10.1145/1978802.1978812

[12] Margaret Ann Francel and Spencer Rugaber. 2001. The value of slicing while debugging. *Science of Computer Programming* (2001), 151–169. https://doi.org/10.1016/S0167-6423(01)00013-2

[13] Robert M. Gagne, Walter W. Wager, Katharine C. Golas, John M. Keller, and James D. Russell. 2007. *Principles of instructional design*. Wiley Online Library. 44–46 pages.

[14] Erich Gamma, Richard Helm, Ralph E. Johnson, John M. Vlissides, and Grady Booch. 1994. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.

[15] Atul Gawande and John Bedford Lloyd. 2010. *The checklist manifesto: How to get things right*. Vol. 200. Metropolitan Books New York.

[16] Julie Gess-Newsome. 1999. *Pedagogical content knowledge: An introduction and orientation*. Springer, Dordrecht. 3–17 pages. https://doi.org/10.1007/0-306-47217-1_1

[17] David J. Gilmore. 1990. Expert programming knowledge: A strategic approach. In *Psychology of Programming*. Elsevier, 223–234. https://doi.org/10.1016/B978-0-12-350772-3.50019-7

[18] Robert M. Grant. 1996. Toward a knowledge-based theory of the firm. *Strategic Management Journal* (1996), 109–122. https://doi.org/10.1002/smj.4250171110

[19] Thomas R.G. Green and R. Navarro. 1995. *Programming plans, imagery, and visual programming*. Springer. 139–144 pages. https://doi.org/10.1007/978-1-5041-2896-4_23

[20] David Hammer and Leema K. Berland. 2013. Confusing claims for data: A critique of common practices for presenting qualitative research on learning. *Journal of the Learning Sciences* 23, 1 (2013), 37–46. https://doi.org/10.1080/10508406.2013.802652

[21] James Herbsleb and Deependra Moitra. 2001. Global software development. *IEEE Software* 18 (2001), 16 – 20. https://doi.org/10.1109/52.914732

[22] R.K. Kavitha and M.S. Irfan Ahmed. 2011. A knowledge management framework for agile software development teams. In *International Conference on Process Automation, Control, and Computing*. IEEE, 1–5. https://doi.org/10.1109/PACC.2011.5978877

[23] Amy J. Ko, Thomas D. LaToza, Stephen Hull, Ellen A. Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching explicit programming strategies to adolescents. In *ACM Technical Symposium on Computer Science Education*. 469–475. https://doi.org/10.1145/3287324.3287371

[24] Thomas D. LaToza, Maryam Arab, Dastyni Loksa, and Amy J. Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25 (2020), 2416–2449. https://doi.org/10.1007/s10664-020-09810-1

[25] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *IEEE/ACM International Conference on Software Engineering*. 492–501. https://doi.org/10.1145/1134285.1134355

[26] Leonard, Dorothy A., Walter Swap, and Garvin Barton. 2014. *Critical knowledge transfer: Tools for managing your company's deep smarts*. Harvard Business Review Press.

[27] Paul Luo Li, Amy J. Ko, and Jiamin Zhu. 2015. What makes a great software engineer?. In *IEEE/ACM International Conference on Software Engineering*. 700–710. https://doi.org/10.1109/ICSE.2015.335

[28] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, problem solving, and self-awareness: effects of explicit guidance. In *ACM Conference on Human Factors in Computing Systems*. 1449–1461. https://doi.org/10.1145/2858036.2858252

[29] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest Q&A site in the west. *ACM Conference on Human Factors in Computing Systems* (2011), 2857–2866.

[30] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest Q&A site in the west. In *ACM Conference on Human Factors in Computing Systems*. 2857–2866. https://doi.org/10.1145/1978942.1979366

[31] Matthew B. Miles and A. Huberman. 1994. Qualitative data analysis: An expanded sourcebook. *Journal of Environmental Psychology* 14 (1994), 336–337. https://doi.org/10.1016/S0272-4944(05)80231-2

[32] Emerson Murphy-Hill, Gail C. Murphy, Joanna McGrenere, et al. 2015. How do users discover new tools in software development and beyond? *Computer-Supported Cooperative Work* 24, 5 (2015), 389–422.

[33] Greg L Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *ACM Conference on International Computing Education Research*. 2–11.

[34] Sirous Panahi, Jason Watson, and Helen Partridge. 2012. Social media and tacit knowledge sharing: Developing a conceptual model. *World academy of science, engineering and technology* 64 (2012), 1095–1102.

[35] Chris Parnin and Christoph Treude. 2011. Measuring API documentation on the web. In *ACM International Workshop on Web 2.0 for Software Engineering*. 25–30. https://doi.org/10.1145/1984701.1984706

[36] Elizabeth Poché, Nishant Jha, Grant Williams, Jazmine Staten, Miles Vesper, and Anas Mahmoud. 2017. Analyzing user comments on YouTube coding tutorial videos. In *IEEE/ACM International Conference on Program Comprehension*. IEEE, 196–206.

[37] Michael Raadt, Richard Watson, and Mark Toleman. 2006. Chick sexing and novice programmers: Explicit instruction of problem solving strategies. In *Australasian Conference on Computing Education*. 55–62.

[38] Arthur S Reber. 1989. *Implicit learning and tacit knowledge*. American Psychological Association. 219 pages.

[39] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How effective developers investigate source code: an exploratory study. *Transactions on Software Engineering* 30, 12 (2004), 889–903. https://doi.org/10.1109/TSE.2004.101

[40] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *IEEE/ACM International Conference on Software Engineering*. 255–265.

[41] Johnny Saldaña. 2009. *The coding manual for qualitative researchers*. Sage Publications Ltd (UK).

[42] Mary Shaw and David Garlan. 1996. *Software architecture: Perspectives on an emerging discipline*. Prentice-Hall, Inc.

[43] Xiaobai Shen. 2005. Developing country perspectives on software: Intellectual property and open source - A case study of Microsoft and Linux in China. *International Journal of IT Standards and Standardization Research* 3 (2005), 21–43. https://doi.org/10.4018/jitsr.2005010102

[44] Leif Singer, Fernando Marques Figueira Filho, and Margaret-Anne D. Storey. 2014. Software engineering at the speed of light: how developers stay current using twitter. In *IEEE/ACM International Conference on Software Engineering*. 211–221. https://doi.org/10.1145/2568225.2568305

[45] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[46] Margaret-Anne D. Storey, Leif Singer, Brendan Cleary, Fernando Marques Figueira Filho, and Alexey Zagalsky. 2014. The (R)evolution of social media in software engineering. In *IEEE Future of Software Engineering*. 100–116. https://doi.org/10.1145/2593882.2593887

[47] Jeffrey Stylos and Brad A. Myers. 2006. Mica: A web-search tool for finding API components and examples. In *IEEE Visual Languages and Human-Centric Computing*. 195–202. https://doi.org/10.1109/VLHCC.2006.32

[48] Visvalingam Suppiah and Manjit Singh Sandhu. 2011. Organisational culture's influence on tacit knowledge sharing behaviour. *Journal of Knowledge Management* 15 (2011), 462–477. https://doi.org/10.1108/13673271111137439

[49] Kyle Thayer, Sarah E Chasins, and Amy J. Ko. 2021. A theory of robust API knowledge. *ACM Transactions on Computing Education* 21, 1 (2021), 1–32.

[50] Christoph Treude and Lars Grammel. 2012. Crowd Documentation : Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow.

[51] Christoph Treude and Margaret-Anne D. Storey. 2011. Effective communication of software development knowledge through community portals. In *ACM Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 91–10. https://doi.org/10.1145/2025113.

2025129

[52] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *International Conference on Software Engineering*. 356–366.

[53] Jaw-Kai Wang, Melanie Ashleigh, and Edgar Meyer. 2006. Knowledge sharing and team trustworthiness: It's all about social ties! *Knowledge Management Research & Practice* (2006), 175–186. https://doi.org/10.1057/palgrave.kmrp.8500098

[54] Douglas Wieringa, Christopher Moore, and Valerie Barnes. 1998. *Procedure writing: Principles and practices*. IEEE.

[55] Jeong-Han Woo, Mark J. Clayton, Robert E. Johnson, Benito E. Flores, and Christopher Ellis. 2004. Dynamic knowledge map: Reusing experts' tacit knowledge in the AEC industry. *Automation in Construction* 13 (2004), 203–207. https://doi.org/10.1016/j.autcon.2003.09.003

[56] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235

[57] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An explicit strategy to scaffold novice program tracing. In *ACM Technical Symposium on Computer Science Education*. 344–349. https://doi.org/10.1145/3159450.3159527