2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)

# An Exploratory Study of Programmers' Analogical Reasoning and Software History Usage During Code Re-Purposing

John Allen
Caitlin Kelleher
johnjallen@wustl.edu
ckelleher@wustl.edu
Washington University in St. Louis
Saint Louis, MO, USA

## ABSTRACT

**Background**: Software development relies on collaborative problem-solving. Understanding previously addressed problems in software is crucial for developers to identify and repurpose functionalities for new problem-solving contexts.

**Objective**: We explore the barriers programmers encounter during code repurposing and investigate how access to historical context about the original developer's goals may affect this process.

**Method**: We present an exploratory study of 16 programmers who completed two code repurposing tasks in different code bases. Participants completed these tasks both with and without access to the historical information of the original developer's goals. We explore how programmers use analogical reasoning to identify and apply existing software artifacts to new goals.

**Results**: We show that programmers often failed to notice analogies, made false analogies, and underestimated the value of reuse. Even when useful analogies were made, programmers struggled to find the relevant code. We also describe the patterns of how participants utilized code histories.

**Conclusion**: We highlight the barriers programmers face in noticing and applying analogies during code reuse. We suggest design recommendations for future tools to allow lightweight evaluation of code to help programmers identify reuse opportunities.

**ACM Reference Format:**
John Allen and Caitlin Kelleher. 2024. An Exploratory Study of Programmers' Analogical Reasoning and Software History Usage During Code Re-Purposing. In *2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3641822.3641864

## 1 INTRODUCTION

Code reuse is an integral part of software development. Not only does reuse allow for more efficient development, it can improve code quality by leveraging weather-tested software components. While recent advances in large language models (LLMs) have rapidly changed the way programmers generate code examples, these snippets are limited in that 1) they are untested, and prone to introduce errors, and 2) they are small in size and complexity.

In 1991, Barnes and Bollinger argued that "good reuse is not the reuse of software per se, but the reuse of human problem solving" [5]. The portfolio of human problem-solving is much larger than what LLMs are currently capable of generating. Programmers reuse the problem-solving of others through opportunistic design, meshing together functionalities of existing software for new purposes [24, 55]. This type of reuse relies on tested, proven code. However, reusing small pieces from non-related projects often leads to code incompatibilities that can be difficult to overcome [17].

Large, existing projects include a wealth of useful software functionalities that are compatible with each other by design. In this paper, we explore *code reuse through re-purposing* in which programmers modify an existing codebase to achieve new high-level goals.

Reusing the problem-solving of other developers presents its own challenges; developers struggle to understand the problems that led to design decisions behind unfamiliar software artifacts [36]. This information is often not readily apparent in either code base nor git repository, so developers prefer to consult with the original developer in order to better understand these artifacts [36]. However, developers today frequently change employers and projects, or are otherwise unavailable, and may leave behind little to no knowledge about their intentions during development.

However, some of this information can be captured as programs are created. As developers solve problems and write code, they often consult numerous resources on the Internet. They search for information and visit official documentation, forums, and other learning sources online [2]. This activity of information-seeking provides a rich context of the problems the original developer has wrestled with through the development process. Currently, we are not aware of any code history tools that include this information. In order to better understand how this information may be helpful, we designed a subgoal-aware code history tool which links developers' stated intentions, web searches, and website visits during development to the code changes they make. In this paper, we explore the process of how programmers attempt to reuse code, and how access to this historical information may impact the process.

In this paper, we frame the process of identifying and re-purposing existing functionalities in a code base to a novel problem as a form of analogical reasoning. We present an exploratory study of 16 programmers completing two code re-purposing tasks: one in the



*CHASE '24, April 14–15, 2024, Lisbon, Portugal*

ACM ISBN 979-8-4007-0533-5/24/04.
https://doi.org/10.1145/3641822.3641864

context of each of two distinct code bases. One task has been constructed to include obvious surface-level analogies between source code and target problem, while the other task's analogies are hidden within the logic of the code itself. We investigate how programmers explore each code base, draw analogies between existing functionalities and their current goals, and make changes to the code bases to implement novel functionalities. We also examine how access to the history of the original developer's stated subgoals and information searches impacts this process.

We address two distinct research questions:

1) Where in the process of analogical reasoning does code repurposing break down?

2) How can historical information about software improve its ability to be repurposed?

Programmers in our study frequently performed surface-level program comprehension, which led to struggles in the analogical reasoning step of *noticing* similarities between their goals and the original developer's goals. Programmers made false analogies and invalid assumptions, which often resulted in "insurmountable" programming barriers[34]. Further, even when programmers did draw correct analogies, they struggled to locate the code responsible for the desired behaviors.

Code histories helped some programmers to *notice* reusable, analogical functionalities by surfacing the sub-problems the original developers solved. Programmers leveraged the histories to link desired functionalities to relevant, concrete code in the code base. Specifically, we identify three usage scenarios in which historical information can be helpful: 1) locating an entry point, 2) scanning the code for relevant changes, and 3) when struggling and stuck.

We use these findings to guide a discussion on how future code history tools should be designed to help programmers identify what problems a code base has already solved, make better analogies between solved problems and novel problems, and locate the relevant code in the software. These features, we argue, will improve the opportunity for "good reuse" to occur.

## 2 BACKGROUND: ANALOGICAL REASONING

In this paper, we consider the process of code *reuse through repurposing* as a form of analogical reasoning, or reasoning that applies the structure of understanding one problem towards solving another problem. Research suggests that people are better at solving "target" problems with access to analogous "source" problems than without [19, 28]. Further, the process of Analogical reasoning tends to be more successful when those analogous problems are very similar to the target problem [8, 28].

Gick and Holyoak proposed that analogical problem solving is a three step process:

- **Noticing** that there is a relationship between the source and target problems. This step is critical and can be supported by prompting [19].
- **Mapping** the parts of the source problem to corresponding parts of the target problem.
- **Applying** the problem map in order to create a solution that is appropriate for the target problem based on the source solution.

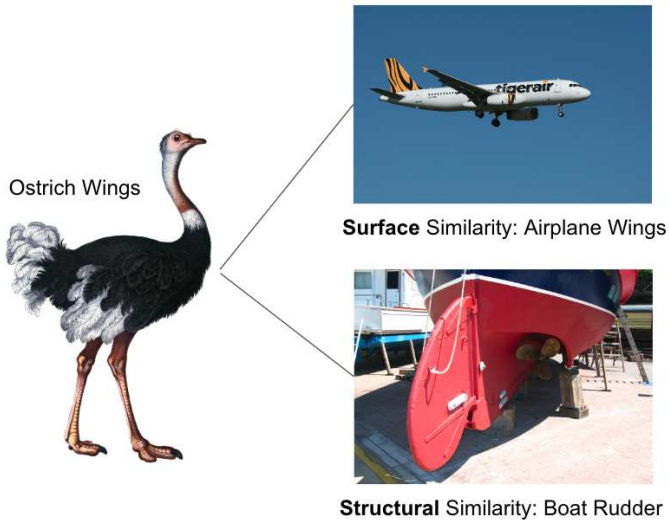


**Figure 1: Surface vs Structural Analogies: While an ostrich's wings [3] are analogically similar at a surface level to airplane wings [56], they are structurally similar to a boat's rudders [48], as they help the ostrich turn the same way a rudder turns a boat. Research shows it is more difficult for people to notice structural similarities than surface level similarities between concepts.**

*2.0.1 Surface vs Structural Analogies.* Research suggests that both the *surface* details and the underlying *structure* of the problem can impact a problem solver's ability to successfully form an analogy between the source and target problems [47]. To demonstrate surface and structural similarity, consider an ostrich's wings. It may be easy to notice the analogy between an ostrich's wings and an airplane's wings, as they appear similar on the surface level. However, it may be more difficult to notice the analogy between an ostrich's wings and a boat's rudder; while they serve the same function, there is not as much surface similarity between the two, as shown in Figure 1. Lacking similar surface details, problems solvers can struggle to notice the correspondence between source and target problems [18]. We replicate this findings within the context of programming, but also identify a new problem: the use false source-target correspondences that are based on incomplete program comprehension.

## 3 RELATED WORK

Our work builds on and contributes to prior research in code reuse, analogical reasoning and programming, and code histories.

### 3.1 Code Reuse

For successful software reuse to occur, the cost of finding and reusing software must be less than the cost of creating it from scratch [4]. Thus, work in this area has focused on program comprehension and code adaptation. Researchers have also explored how to design code to encourage and support subsequent reuse.

*3.1.1 Program Comprehension.* Before reusing code, programmers must understand the code, a process that is often inefficient [33] and

cognitively overwhelming [12, 40]. During program comprehension, programmers inefficiently browse information [34, 58] while attempting to answer questions about the code base [14, 49]. In particular, programmers find it hard to understand the reasoning for the implementation decisions of the original developer [33, 36, 37].

Programmers may employ some combination of bottom-up or top-down strategies when understanding a program's behavior [38, 39]. Bottom-up strategies involve deducing program behaviors by looking at lines of the lowest level of code in order to build up their understanding of the software [7, 46]. Top-down strategies generally consist of programmers mapping high-level steps of the application domain to functionalities within the software [6, 51]. Often, programmers begin with a top-down approach in order to find specific functional implementations, and then rely on a bottom-up approach to understand the implementation itself [11].

*3.1.2 Re-purposing Code.* Once programmers have an understanding of the source code base, they must begin to reason about connections between the source and target problems [5, 13, 50], and their corresponding code bases [44].

While past work has identified problems that programmers encounter during reuse, little research explores the process of code reuse. Some work shows that developers may reuse large portions of software projects as "templates" for new projects [43]. Haefliger [22] investigates the characteristics of API and library based reuse by open-source developers using a combination of interviews and code analysis over time. Other work in this area also evaluates open-source developers' likelihood of reusing code based on their attitudes and organizational characteristics through code analysis [44] and survey data [50]. Code analysis is able to analyze a large amount of reuse cases, and has shown that code reuse can lead to increased productivity, but may also introduce bugs [44]. Surveys show that efficient reuse is difficult when trying to solve challenging problems [50].

Much of this existing work is based on retrospective research, and, consequently, does not provide a full picture of the process of reuse. Further, the existing literature has tended to focus more on reuse via libraries and APIs as it is easier to identify in a code base. Our study focused on the process of software reuse via re-purposing. Using a lens of analogical reasoning, we identify challenges that arise through the reuse process.

*3.1.3 Designing to Support Reuse.* The final branch of research in this area focuses on designing software that is more readily reusable. To date, most code reuse research focuses on the importance of designing software to be reusable [5, 29, 42], including designing modular software [5, 42], providing adequate documentation [16], and reducing costs of searching relevant code [31]. Our work contributes insights into the needs of secondary programmers.

## 3.2 Analogical Reasoning and Programming

Fundamentally, code reuse requires programmers to reason about the connections between code they have access to and problems they are trying to solve [5, 13, 50]. This process can be seen as a form of analogical reasoning [23]. Research in analogical reasoning suggests that people are more successful in reusing solutions from analogous problems that are more similar to their target problem [8, 28]. Similarly, Krueger [35] suggests effective code reuse practices should minimize the cognitive gap between the original idea of a system and its eventual executable implementation. One study found that programmers were better able to recognize abstract analogies, but more able to use concrete analogies [53]. Generally, research around analogical reasoning and programming falls into two groups: 1) exploring the relationship between analogical reasoning and programming skill and 2) proposing tools that leverage analogical reasoning to support reuse.

Some existing research around analogical reasoning and programming has explored the degree to which the underlying skills are related. Research suggests that analogical reasoning ability predicts novice programming ability [9] but programming instruction does not consistently improve analogical reasoning [21] [54].

Some work proposes using analogical reasoning as a basis for reuse tools and works towards the technical ability to build such tools [20, 52]. Neither of these tools were directly integrated into a programming environment or evaluated with programmers.

While prior work has recognized the relationship between analogical reasoning and programming, we are not aware of prior work that has explored actual code reuse tasks through the lens of analogical reasoning. This paper contributes by 1) identifying several conditions that lead to programmers failing to make meaningful analogies between their goals and source code they have access to, 2) describing barriers programmers face after identifying useful analogies, and 3) proposing design guidelines for future tools to improve programmers' ability to find and use relevant analogies in source code.

## 3.3 Code Histories

Research suggests that history information is valuable to developers during program comprehension and that current ways to capture it are insufficient.

LaToza and Myers [36] surveyed software developers about the questions they typically have about code that are difficult to answer. They found that the most commonly reported hard-to-answer questions involved the design decisions behind the code, which are typically not documented in a code base [36]. Ko et al corroborates this finding when observing developers as they worked [33].

Today, historical information for a code base typically comes through versioning control. Research suggests that git commits are not enough to answer developers' questions and highlight several issues. Developers often incorporate multiple unrelated changes into a single commit [26, 27, 32]. These "tangled commits" can make it difficult for programmers to accurately determine the reasoning for particular code changes [25, 27]. In response, some work explores capturing more detailed code histories such as keystroke level data [45, 59, 60]. However, keystroke level data, while arguably complete, contains a lot of information that is not helpful, leading to a new challenge: extracting relevant information [40].

We are unaware of any work that explores programmer usage of code history tools that include the information-seeking web activities of the original developer, or includes subgoal labels at this level of granularity. In addition, our work contributes an initial exploration of how code history information can be used during a code reuse process.

# 4 METHODS

To better understand the reuse through re-purposing process and the potential impact of code history information, we conducted an exploratory study of sixteen student-programmers completing two code re-purposing tasks in a lab setting, with and without access to code history information. Tasks were designed to represent a breadth of reuse scenarios, and capture two phases of code reuse: 1) identifying which functionalities can be reused, and 2) mapping desired functionalities to the source code responsible. For one task, participants had access to a web page including historical information about the related code base.

## 4.1 Participants

We recruited sixteen participants via a university e-mail list including both graduate and undergraduate students in computing based degree programs at a private university. To ensure that participants had adequate computing backgrounds to make progress on reuse through re-purposing tasks, we recruited only students who had taken at least four programming classes or who had at least three months of work or internship experience programming. In practice, our participants had significant experience, reporting an average of more than nine computing courses and more than sixteen months of work experience. Twelve participants were enrolled in undergraduate programs, and one was concurrently pursuing a Master's. The remaining four participants were pursuing their PhD. All participants were in degree programs focused on either Computer Science or Computer Engineering.

## 4.2 Study Procedures

Participants met in person with the researcher in a one-on-one lab setting, and completed two reuse tasks. For one of their two tasks, participants had access to a code history. To control for the effects of task order and story access order, we employed a within-subjects Latin-squares design to balance the possible combinations of task and story access order.

A researcher began each session by briefly describing the participant's first task and instructing the participant to think-aloud while working. Participants then had thirty minutes to complete the task. At either the end of the thirty minutes or upon task completion, the researcher conducted a brief semi-structured interview. The interview focused on understanding the participant's specific strategies and knowledge gaps. The second task followed an identical structure. On the task in which the user was given code story access, the researcher opened the code story in a web browser, briefly described its purpose, and demonstrated how to browse through the history information.

*4.2.1 Data Collection.* During each user test, we logged the web searches and websites each participant visited, the text of all code files each time participants saved them, and the times when participants tested their code or consulted the code history. In addition, we collected screen and audio recordings of participants' programming tasks and interviews, as well as high-level field notes of perceived or stated activities during each test. All interviews were transcribed and used for thematic analysis and behavior identification.

## 4.3 Thematic Analysis

To extract themes from participants' utterances either while working on a task or during the interview, we first transcribed the audio from each user test. One author created a set of quotes for analysis by excluding utterances that focused on building rapport, contained incomplete thoughts, consisted of reading something on screen aloud, focused on syntax, or were unrelated to the reuse task. To uncover themes in the quotes, we analyzed the quotes in three distinct passes by 1) reading through them as a team 2) discussing emergent themes and 3) assigning quotes to groups based on the themes that emerged. These themes are not directly included in our results, but were used to identify routes of investigation.

## 4.4 Labeling Participant Strategies

The research team summarized each user test into discernible goals and strategies participants took that could be mapped to concrete actions such as code changes, web searches, or verbalized quotes. For example, programmers' "attempted strategies" are based on either 1) think-aloud verbalization during the task, or 2) tangible code additions paired with confirmatory post-task interview questions. We then counted each time these identified strategies were pursued by programmers.

# 5 EXPERIMENTAL TASK DESIGN

We designed two tasks which require participants to re-purpose elements from a given code base, ensuring that each code base represented a real-world project that was developed in an authentic way. We presented two tasks in order to capture a wider breadth of code reuse scenarios, and insights across the range of the steps of analogical reasoning.

In particular, we designed one task to include deep **structural** similarities that are not immediately obvious, and the other task to contain visually obvious, high **surface** similarity analogies with the provided code base.

This dichotomy is intended to highlight barriers within the specific steps of the analogical reasoning process, from 1) *noticing* similarity between target and source problems, to 2) *mapping* and *applying* source code solutions to a target problem.

## 5.1 Structural Analogy Task: Image Processing Project

The *structural* analogy task is intended to identify barriers during programmers' processes of *noticing* analogies between the source code and a novel problem. This task requires participants to solve an image processing task when given access to a code base that already addresses many related image processing problems, but has few surface-level parallels.

*5.1.1 Code base: Photo Mosaic Generator.* The Photo Mosaic Generator code base is a program that takes a target image and creates a photo mosaic of that target image using a library of images that can be part of the mosaic. The implementation is written in Python in a modular style, with functions that handle specific sub-problems.

The Photo Mosaic Generator code base takes two inputs: a `target_photo` and a `Photos/` directory, which includes several

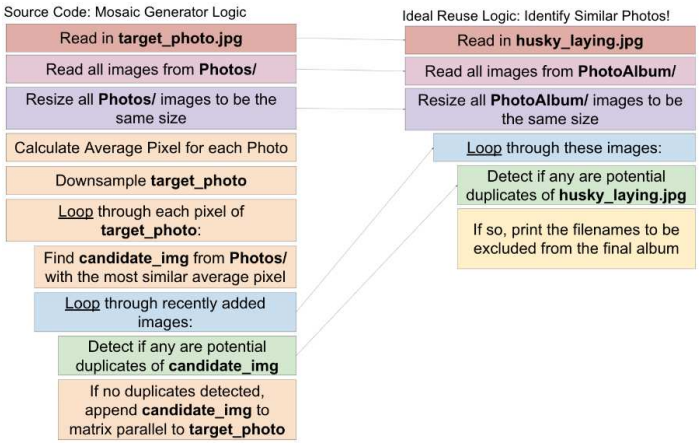


**Figure 2: The "ideal" reuse scenario for the structural analogy task includes participants reusing this logic from the Mosaic Generator.**

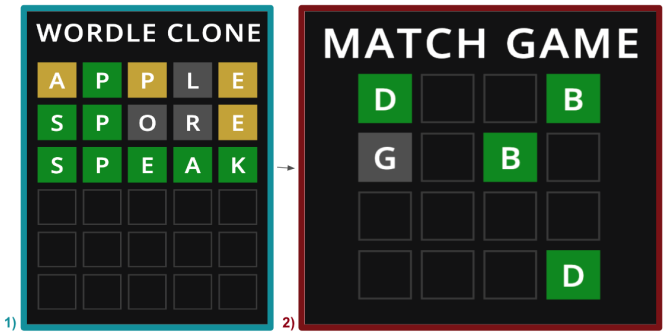


**Figure 3: Memory Matching Game: This task requires participants to transform a Wordle clone (1) into a Memory Matching game (2). The matching game visually shares many surface features to the original Wordle game.**

hundred images. The code uses these inputs to generate a mosaic of the `target_photo` with pictures from `Photos/` (see Figure 2).

*5.1.2 Experimental Task: Identify Similar Photos!* The Photo Mosaic Generator reuse task challenged participants to create a console application to identify highly-similar images, so that they can be excluded from a photo album. As with the original program, the target program should take a source image and then a directory of images to scan for similar photographs. The output should be a list of photographs that are similar to the source image.

*5.1.3 Ideal Reuse Strategy: Identify Similar Photos!* The functionalities to complete this task are already implemented in the Photo Mosaic code base. Figure 2 shows how the task can be implemented using functionalities from the Mosaic Generator code.

*5.1.4 History Capture: Photo Mosaic Generator.* This project was developed by a member of the research team. To capture the history, we used a history capture tool we developed that is integrated into VSCode and Chrome. Instead of manually labeling each action and appending it to a database, the capture tool saved code changes, web visits, and output automatically.

## 5.2 Surface Analogy Task: Tile Games

The *surface* analogy task is designed to identify barriers after the programmer "notices" an analogy. This task requires participants to create a tile-matching game when given access to another tile-based game. In this task, the reuse analogy is easy to identify, but the program is sufficiently complex such that finding and reusing specific program features will require more effort.

*5.2.1 Code Overview: Wordle Clone.* The code base for this task is a simple implementation of the online word game, "Wordle". The code is written in HTML, JavaScript, and SASS, and is compiled using Gulp, a system that automates build tasks in web development. In the code base, Gulp [41] is used to minify the Javascript and compile SASS to CSS.

While the structure of the game tiles (shown in Figure 3-1 is created by `index.html`, the general interactivity logic of the Wordle clone, in `app/js/script.js`, is as follows:

- A) Choose random word from list of possible words
- B) When user types a letter, display the letter
- C) When user "Enters" a five-letter word, update tile styling to give information about the presence of each guessed letter in the solution word

The JavaScript file consists largely of helper functions which are integrated into the logic of a main "keydown" listener, which controls the flow of logic for the program.

*5.2.2 Experimental Task: Memory Matching Game.* The high level reuse through re-purposing task for the Wordle Clone code base was to create a matching game. Programmers needed to create a memory tile-matching game in which they have a 4x4 grid of tiles, with two tiles each assigned with the first 8 letters of the alphabet at random. The tiles should begin facing down, with their letters hidden from the player. When the player clicks a tile, it should flip over, revealing the letter on the tile. When the player clicks a second tile, the second tile should also flip. If the two revealed letters match, then they stay in the flipped state. If they do not match, both letters flip back over and the player tries again. This process continues until all tiles are matched together. The original and target program outputs are shown in Figure 3.

*5.2.3 Ideal Reuse Strategy: Wordle Clone.* The Memory Matching task requires locating and understanding the code implementation of desired behaviors. Programmers need to 1) determine where game elements and interactivity are defined, and 2) figure out how the different files (JS, HTML, CSS) communicate to implement these behaviors. To complete the task, participants should:

- Find and edit the HTML responsible for the Wordle clone's 5X7 grid, and transform it into a 4X4 grid.
- Locate the Wordle event-driven structure and change the main function to be triggered upon a "click" event, rather than a "keydown" event.
- Reference HTML tiles from the JavaScript side.
- Create new logic for the memory matching game.

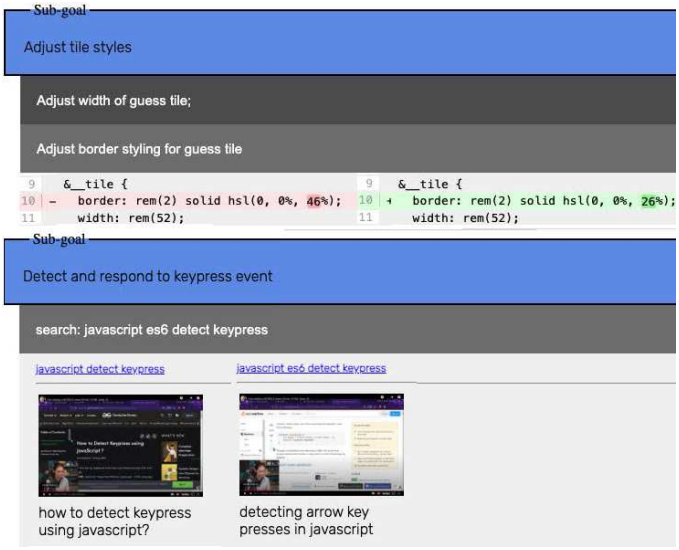


**Figure 4: Code history displayed as an interactive webpage that features collapsible items nested within discretely labeled subgoals of the original developer.**

*5.2.4 History Capture: Wordle Clone.* YouTube now contains a subcommunity of developers who record videos of themselves coding. The Wordle Clone code was originally created by the popular code streamer "Coder Coder" [10]. Coder Coder recorded her process in a nearly five-hour YouTube video, during which she develops this code for the first time in a raw, unedited recording while describing to her viewers her current goals and challenges at any given time. Our team watched the video, captured all code changes, web searches, web visits, and organized them into developer goals.

## 5.3 Code History Implementation

The thought processes behind a programmer's design choices are often unclear [33, 36], as programmers often attempt and fail various alternative approaches before implementing their ultimate solution [57]. In order to explore how programmers referenced this information during code reuse, we created a code history tool which reveals all the sub-problems and solution attempts the original developer had during the development phase.

For one of their two experimental tasks, participants had access to the code history corresponding to their source code base. These code histories were presented through a webpage (see Figure 4) and featured a list of subgoals the original developer pursued while building the relevant code base. Each subgoal could be expanded to reveal activities related to that subgoal, including 1) diffs of changes to modified code, and 2) thumbnail links for web pages that the original developer visited while working on that subgoal.

# 6 RESULTS

We address two distinct research questions:

- 1) Where in the process of analogical reasoning does code re-purposing break down?
- 2) How can historical information about software improve its ability to be repurposed?

We find that programmers struggle in **noticing** re-usability of structurally-analogical functions that lack surface-level analogical similarities to their goals, leading them to make false analogies with the code and attempt to reuse components that are not helpful to them. When programmers do notice analogical functionalities, they struggle to identify and modify the code responsible for the desired behavior. Additionally, we find that participants underestimated the value of reuse when analogies were not immediately obvious, and did not always invest time in trying to identify analogies between their problem and those which the program had already solved.

Second, we find that code histories were helpful when used, and identified specific scenarios in which history information was valuable for programmers: 1) locating an entry point, 2) scanning existing code, and 3) when stuck or struggling.

## 6.1 Structural Analogy Reuse: distinguishing relevant functionalities

The first step in successful code reuse is distinguishing which functionalities of a program can be reused [35]. Similarly, the first step in analogical reasoning is "noticing" the analogy between a source problem and a target problem [28]. We found that programmers often failed to make ideal analogies, and thus never "noticed" the reusability of useful functionalities. This arose in two ways: failing to select existing code that minimized new development and creating analogies based on incorrect understanding of code behavior.

*6.1.1 Shallow program comprehension led programmers to miss reuse opportunities.* The structural analogy reuse task focused on identifying similar photos by re-purposing a program for creating photo mosaics. In the research team's opinion, the *ideal* way to approach the task is to reuse the existing logic that 1) reads in and resizes the images, and then 2) to compare the similarity of images. Hereafter, we refer to this as the *ideal* approach. Using the ideal approach, a programmer would only need to loop through the images and print their file names if they are marked as "similar" to the target image, this process is shown in Figure 2. Participants almost never holistically studied the code before settling on a strategy, and thus rarely identified the ideal reuse scenario.

The majority of participants developed an initial plan to solve the problem and did not revise it throughout the duration of the task. Participants typically identified one "anchoring" method that they could envision as part of the solution and attempted to develop a solution based on this anchor. Our observations suggest participants' plans were based on a shallow understanding of the code largely based on method names. For example, after User 4 stated "When I started looking at [the code], and for the first few minutes, definitely I was like, Oh my lord, like, what does this even mean?". At the end of the period User 4 describes, they found the `get_avg_pixel` method as relevant, and ignored all other helper functions. When asked how they selected what code to try to reuse, User 4 responded by admitting "I kind of just stuck to like what I decided my goal was, which was [to] compare the average pixels."

To successfully use the majority of the program's methods, users needed to be able to load and scale the images. Thirteen participants (81.25%) reused code to load images into memory. Of the remaining 3 participants, one made no code changes, and the other two attempted to load images from scratch.

Only six of the sixteen users (37.5%) correctly identified the functionality for image similarity as an important element to reuse. Of those, three (18.75%) were able to fully complete an implementation following the ideal approach. It is worth noting that these were the only three participants to complete the task, resulting in an 18.8% overall success rate.

*6.1.2 Programmers created false analogies based on incorrect understanding of code, and stuck with them.* Eight of our sixteen participants (50%) attempted strategies based on false analogies during the structural analogy task. We define a false analogy as one in which the programmer has an incorrect interpretation of how a given section of code relates to their current problem, and uses it as the basis for a solution plan. Because the underlying assumptions about the code's behavior are incorrect, any plans the programmer created based on these assumptions were consistently invalid as well. The plans arose when participants assumed code behavior based on surface features, often method names.

Five of sixteen participants (31.25%) created a plan based around reusing the code for comparing the similarity of a pixel within a photograph to the pixel averages of available photographs in the image set. User 14 formed an initial plan around comparing pixel averages. They quickly calculated the average pixel value for the target image and each of the comparison photos. When they felt stuck, they turned to the code history and found a subgoal relating to comparing image distance (the function used in the ideal approach). They briefly experimented with this code, but did not pass the right parameters. They quickly reverted back to their original and ill-fated average pixel approach.

Three participants identified the code for using KMeans to perform clustering. These participants assumed, incorrectly, that KMeans was clustering images. In fact, KMeans was used to cluster the pixels within a given image. The participants in this group verbalized plans to cluster the images, and then identify duplicate images based on the cluster groupings produced. This is not a feasible solution, since there is no way to ensure Kmeans clusters will represent duplicate or near-duplicate images. User 2 attempted this approach, and successfully printed clusters of pixels for each of the comparison photos but then was unsure of how to proceed. "I don't necessarily know how to compare one set of RGB to another set of RGB now perhaps there's a method I've forgotten about so I'm gonna go look for that. I know there's some comparison methods". Along the way User 2 notices and rejects the ideal path. "I can compare image to image, like I can [call] `compare_image_distance()` but to do that now I don't think that works based on what I've built up so far." They reluctantly continued their approach, later reflecting "The reason I didn't want to do that is because that will scale horribly."

In summary, participants made false assumptions that led to invalid reuse strategies, and were reluctant to switch strategies even after finding a more promising approach.

## 6.2 Surface Analogical Reuse: identifying relevant code

During the surface analogical reuse task, participants successfully identified analogies between Wordle and a Memory Matching game, but encountered three barriers: 1) locating interface elements in code, 2) testing changes, and 3) adapting method calls.

*6.2.1 Locating Interface Elements in Code.* Participants overwhelmingly began the Wordle clone task by attempting to find where the tiles are created. Since the tiles contain no characters initially, searching for interface text is not helpful. Instead, some participants used a keyword search for numbers matching the dimensions of the existing board. This led some participants to irrelevant, hard-coded game logic. When they modified these numbers, they were unable to observe changes in the output. Others unsuccessfully browsed the JavaScript and SCSS files looking for the tiles.

User 3 exemplifies a mixture of these approaches. He began with the goal of converting the 5X7 Wordle grid to a 4X4 grid, and immediately focused on `script.js`, which handles the interaction of the Wordle site. He noticed that there are "for loops" that iterate five times. Although the loops handle game logic, he believed, based on the number of iterations, that these loops generated the tile layout. Accordingly, he modified the loop to iterate 4 times, but observed no output changes. As in the structural reuse task, a shallow understanding of the code led to false assumptions and an unsuccessful solution plan.

*6.2.2 Testing Changes.* In our surface analogy reuse task, participants sometimes struggled to evaluate whether their changes were moving in a productive direction. Participants faced some challenges with the project's build structure and the web context. At the core, these challenges were related to knowing which code was going to be called and where to find its output.

After editing the for loops and not seeing output changes, User 3 next attempted to print to the console. He checked the terminal console rather than the browser console and thus did not see any feedback. He then located `dist/script.js`, a minified version of the script intended for web distribution. It is formatted for efficiency and is not intended to be human readable. Yet, User 3 tried to reformat it in the hopes of finding a solution starting point.

The Wordle project used a common build structure in which the local javascript files are compiled into a single distribution file for efficiency. Five participants (31%) had issues involving the `dist/` files. Three of these participants attempted to manually reformat the `dist/script.js` file to increase readability. This was a tedious process that took an average of over 10 minutes.

User 16 exemplifies the frustration programmers felt trying to trace through minified code, stating "That's where I got stuck. Because the [script] that the HTML was calling has only one line [of minimized code] where apparently everything is being inserted. Yeah, so I just decided I'll just do it from the scratch instead of me trying to figure it out what exactly is going on."

*6.2.3 Adapting Method Calls.* While attempting to modify the Wordle code base into a matching game, participants encountered situations where they needed something similar to a method call in the source code base, but struggled to adapt the existing method call to suit their needs. For example, this arose in the context of reusing a

JQuery call similar to one created by the original developer. Participants struggled to understand the naming conventions the original developer applied to access the tiles and were therefore unable to adapt the needed JQuery call. Only half (eight participants) were able to adapt the JQuery call for their new use case.

## 6.3 Use of Code History

The code histories were used rarely; only half of our participants ever referenced them. However, when used, participants benefited substantially from viewing the Code History. We observed that history information was relevant when participants were: 1) locating an entry point, 2) scanning existing code, and 3) struggling.

*6.3.1 Locating an Entry Point.* The provided code histories were best suited for a top-down style of problem solving in which participants could use them at the beginning to find a good starting point for their current subgoal.

User 15 illustrates one successful use of code history to find an entry point into the code. User 15 began the structural analogy reuse task by scrolling through the Photo Mosaic code, noting that "this is a lot of functions". Perhaps overwhelmed, they turned to and read through the code history. User 15 noticed the subgoal associated with ensuring similar-looking images are not placed adjacently in photo mosaic. They clicked this subgoal and studied the associated code added by the original developer. Because the subgoal provided high confidence that this section of code was relevant, User 15 began trying to understand it. 'I think this should give me a number." they stated while highlighting the line of code responsible for determining if photos are "similar" ("if mse < 90:"). "And if they're similar, it should print the file". User 15 then copied the code into their code base and quickly completed the task.

*6.3.2 Scanning Existing Code.* One of the strategies participants employed for both tasks was reading and scanning code. Participants did not refer to the code histories while scanning existing code, but their behavior and struggles suggest it represents an opportunity to integrate history information. When faced with reading an entire code base, even one of modest size, participants were reluctant to commit to attempting to fully comprehend the code. Instead, they relied on information scent cues in order to determine whether code was likely to be helpful. This was a greedy process in which participants often picked an initial strategy and stuck with it, even when it was not going well, as described in Section 6.1. Yet, the subgoal information contained in the code history for the sections of code participants scanned could have provided another form of low cognitive cost evaluation that may have helped to steer them towards more relevant code sections.

*6.3.3 Struggling.* Participants who referred to the code histories when struggling were largely successful, although the successful strategies of use were different for the structural and surface analogy reuse tasks.

Participants working on the structural analogy reuse task most frequently turned to the code history when they couldn't make sense of the source code. The most successful uses were ones in which the participant browsed the history with no particular goal in mind. This encouraged them to browse through abstracted subgoals as opposed to raw code, exposing them to an overall sense of what the developer worked on, and giving them ideas on where to start. Four programmers identified reuse strategies when browsing the history without any explicit information goal.

In contrast, users working on the surface analogy reuse task turned to the code history when they struggled to find the implementation of a particular element. These uses were most successful when programmers had a specific information need in mind. Four users (25%) searched for information in the surface analogy task. All of these code history uses were successful, revealing needed information including the locations of elements of interest such as the tile definitions or a needed event-handler.

For example, User 5 turned to the code history after failing to find where the tiles were created using the grid size. In reading through the code history, User 5 found a subgoal related to defining tiles, and expanded it, revealing modifications to `index.html`. When asked about it later, he stated "The HTML file took me a while to find, because I was looking for this, but I was looking in the wrong place... I figured it would be somewhere in the... Because I think CSS is like the look of the site, so I figured it would be there, but it was in the HTML."

## 6.4 Underestimating the Value of Reuse

Research shows that programmers will often choose to re-implement a given functionality over trying to reuse it [4]. In our study, domain expertise appeared to play a role in this decision.

User 3 went straight to attempting to implement the image similarity task from scratch, without ever consulting Photo Mosaic code, or its history. When asked why, the participant cited a Master's degree in statistics and said "So I'm almost certain that this cross correlation is the best or close to the best way out of the box to do [image similarity]. If you showed me something I've literally never seen, I'm gonna read documentation. But if I have a really good idea of how to do it, I'm just gonna go ahead and try my best." It is worth noting that User 3 attempted to re-solve multiple sub-problems completed by the original developer, such as loading images into Python and resizing them, but did not complete these sub-goals during the allotted time.

In contrast, User 10 who had not worked with images before, stated "I have to reuse [the code] because off the top of my head, I don't know how to write code to analyze images. So this is the part that I'm gonna have to pull from this Photo Mosaic code." User 10 ended up successfully reusing code to read in and compare image similarities, while neither of the two programmers who attempted the solution from scratch did.

This represents an interesting design challenge going forward. Even for programmers with significant domain expertise, adaptation has the potential to be more efficient than re-implementation. Integrating reuse support into developer's natural workflow better may help to highlight reuse opportunities for domain experts.

# 7 DISCUSSION AND DESIGN GUIDELINES

Our findings highlight several opportunities for code histories to improve the code reuse process. We describe design guidelines for future code history tools based on our study.

We argue that code history tools should 1) be incorporated into the natural workflow of developers, 2) allow lightweight evaluation

of code by linking code segments to the relevant subgoals of the original developer, 3) clearly mark historical code that is no longer present in the final code, and 4) support noticing reuse opportunities through search.

## 7.1 Integrating Code History Tools into the IDE

While the code histories were lightly used during this study, the usages we observed were largely successful. The sub-goal information and its connection to specific code helped programmers to make higher confidence assessments of code relevance. When programmers were more confident that a given segment of code was relevant, they also appeared more willing to invest effort into program comprehension. However, the code histories were not well integrated into programmers' natural workflow. For example, code history information might enable programmers to make more accurate assessments of code function when searching for potentially relevant code. Since the code history information was presented in a separate web page, programmers were not able to leverage it in their programming context. From a cognitive load perspective, the separate web page may have introduced extraneous cognitive load, dividing programmers' attention and increasing the cost of use.

We suspect that some of the light usage may also have been due to lack of familiarity. Programmers already had established habits for exploring unfamiliar code bases. We introduced programmers to code histories but did not ask them to complete an introductory task using a code history. Future studies should include warm-up tasks that more fully introduce code histories. Several of the participants who explored the code histories commented about how useful the information was. As user 13 reflected, "Looking at the code history was useful as a supplement to the actual code. It was telling me what they (the original developer) were thinking at that time.. Like, is it similar to what *I'm* thinking? That's actually how I found the [game] tiles, looking at the code history.. I wish I'd looked more in depth at the code story. "Going forward, our results point to opportunities to introduce code history information within the programming environment to align with the reuse strategies we observed.

## 7.2 Enable quick evaluation of software by linking code with relevant subgoals

Participants in our study were frequently overwhelmed by the amount of irrelevant code they had to sift through in order to find relevant functionalities. However, the irrelevant code often included useful information that standalone snippets do not.

As User 1 stated, "Having the irrelevant code is nice in that sometimes it can have like [usage] snippets [of the desired functionality], but at the same time, I found it really noisy and I find it really hard to concentrate on the parts that I actually wanted."

Code history tools should allow for lightweight evaluation of a codebase by maintaining a two-way connection between abstract subgoals of the original developer(s) and code elements. The subgoal descriptions provide another source of surface features programmers can easily evaluate, as well as a pathway to get more details relevant to them. As we saw among the code history users, the availability of subgoals would support programmers in two of the three history usage scenarios we identified: locating an entry point and scanning existing code.

*7.2.1 Connecting subgoals to code.* Code history tools should support programmers attempting to *locate an entry point* by allowing programmers to scan through high-level logic and jump into the code where they deem appropriate. In the **surface** analogy task, participants easily identified reusable functionalities, such as tile flipping and the board's design, but struggled to find the relevant underlying software elements in the code base. Four participants (50% of those with Wordle history access) were able to use the code history to identify which files were edited to contribute to specific functionalities, however, this process could be improved by tools that directly link historical changes to their existing implementations in the code. This design goal is in-line with Krueger's recommendation that reuse is improved through abstraction [35]; by linking code elements to the abstract subgoals of the original developer, history tools can improve future programmer's ability to reason about abstract functionalities to be reused.

*7.2.2 Connecting code to subgoals.* Code history tools should expose relevant subgoals for code segments to enable programmers to make more accurate assessments of code function.

Digging into an unfamiliar code base is a cognitively demanding task, one that multiple participants indicated was overwhelming at times. As User 5 described the matching game task, "There was like no way I could have done that. I have done matching in Java, but it was with like my own objects. I had like an object setting up the board, an object that divides the color, but this [code base] had nothing like that."

Perhaps in response to the cognitive demands, participants seemed to scan the code for a shallow understanding of the behavior of given segments of the code, using cues that required little cognitive effort such as method names and comments to hypothesize the code's function, often leading to incorrect deductions about code function.

Further, participants used a greedy approach to finding a reuse plan, often building on the first method they found that seemed promising. Once programmers had begun to implement a plan based on a non-ideal starting code selection, they were hesitant to change course, even when they found other code that seemed more promising. This behavior is consistent with the sunk cost fallacy. Taken together, these two tendencies suggest that it is important to support accurate lightweight evaluation of unfamiliar code.

## 7.3 Identifying removed code

Code history tools should clearly indicate when historical code additions are longer incorporated into a code base, and point to existing code that is related to the deleted code whenever possible. Two participants found code additions in the software history that they found relevant, but were no longer included in the final code. This introduced a few issues. First, participants invested more time in the hopeless search of trying to find the lines of code they were interested in within the final code. Second, participants were trying to use code snippets that had been abandoned by the original developer, which has the potential to lead to errors down the road. While *linking subgoals to code* may help prevent these errors, we

suggest that code history tools also explicitly denote changes that are no longer included in the present code state, and point to the moment in the history where they are ultimately discarded. This may help the user identify how the original developer addressed the same functionality with a different, more preferred strategy.

### 7.4 Using search to support for noticing analogies

Code history tools should detect when programmers make similar searches as the original developer and help them find related code that is already integrated into the program.

As a result of not fully comprehending the code base, programmers struggled to notice functional analogies between existing code, and code they wanted to write. During the **structural** analogy task, we frequently observed participants attempt to re-solve problems that had already been solved in the Photo Mosaic code base (7 (43.8%) participants).

When doing this, participants often made similar searches, and visited websites with semantically similar titles as the original developer. This happened for two reasons. Users used web resources to better understand what the code was doing, and second, participants turned to web resources in their own attempt to solve a problem that the original developer already worked on. In either scenario, identifying searches similar to those in the history may provide another entry point that helps programmers to make use of existing codebases.

For example, User 4 attempted to add key listeners dynamically when the page loaded during the Structural Task. In doing so, he searched for "document load javascript". Unsatisfied with the results, re-framed his search to "document load javascript event" and ultimately "document load javascript event DOM". Eventually, he found a relevant example and integrated it into the code base.

However, the original Wordle developer of the code base had already added a similar functionality. When designing the random letter selection, the original developer searched "javascript run function on page load", before adding code to her script to generate a new random word each time the page was loaded. In retrospect, this was a potential point at which a codebase-grounded search result could have directed the user towards 1) the relevant resource the original developer referenced when working on the same problem, and 2) the code the original developer created after finding said resource. Future code history tools should consider analyzing developer searches as an entry point to present relevant history.

## 8 LIMITATIONS

Internal threats to validity include the task time constraints, and the potential for programmers to gain experience with code reuse between the first and second tasks. The Latin-squares design minimizes the impact of any experience effect by evenly distributing task orders. We note that the time limitation may have encouraged programmers to continue pursuing unsuccessful strategies, due to the potential time cost of re-starting. Finally, a lack of familiarity with the history tool may have led to its light usage.

External threats to validity include the small and narrow participant pool and the limited set of reuse tasks. Our task selection process was designed to get participants to recycle real-world projects, does not account for all reuse scenarios. For example, some forms of code reuse involve the step of finding and selecting reusable code. We removed this step in order to focus on the analogical alignment and code modification aspects of reuse. Future work should include a broader subject pool and a wider variety of tasks.

## 9 FUTURE WORK

There is relatively little work exploring the process of reuse and even less exploring how code history can support reuse. We believe that future work should concentrate on: 1) broadening research combining analogical reasoning and code reuse, 2) designing and evaluating code histories that integrate into the workflow of code comprehension and reuse tasks in a variety of domains, and 3) exploring techniques for fully automatic code history generation.

### 9.1 Analogical Reasoning and Code Reuse

Our study explored reuse through repurposing, but analogical reasoning is a potentially useful lens through which to consider a broader cross section of code reuse activities. In our study, analogical reasoning helped highlight the problem of false analogies due to incomplete program comprehension. Future studies should consider other types of code reuse through the lens of analogical reasoning to generalize these results and identify differences in programmers' needs based on reuse type.

### 9.2 Integrating code histories with programmers' workflow

The code histories contained information that was helpful to participants during reuse tasks. However, utilization remained low, perhaps due to unfamiliarity and lack of integration with programmers' workflow. Future research should further characterize programmers processes across diverse reuse contexts to identify where and how these tasks may be supported using code history information. While our study focused on the reuse process, code histories may also be be helpful in helping programmers to evaluate the potential utility of code when making decisions about reuse.

### 9.3 Automatic code history generation

While our current code histories are created in a semi-automatic process, they still require hand annotation of the subgoals and activities. It is well documented that developers do not wish to spend more time documenting their code [1, 15], and do not want to invest much time in explaining their intentions while writing code [30]. As such, tools should investigate capturing and annotating this information passively. Future work should explore techniques to fully automate the capture of code history information, potentially by leveraging Large Language Models (LLMs).

## 10 CONCLUSION

Our study suggests that code reuse can be limited by programmers' tendency to use incomplete program comprehension while planning their reuse strategy. Programmers' surface-level understandings of code led to failures to notice analogies between code they had access to, and problems they were trying to solve. In our study, this resulted in participants making sub-optimal analogies

and pursuing reuse strategies that had little merit. Further, once programmers selected an approach, they were unlikely to switch to a new one, even after noticing functionalities more aligned with their original goals. Programmers also struggled mapping desired functionalities to their actual software implementations.

Programmers' uses of the code history suggest that history information can support programmers' in making lightweight evaluations of program function. This support is relevant to the primary barriers our study identified for locating reusable software components and their code implementations. However, code histories need to be more tightly integrated with developers workflow patterns. Specifically, code histories should support developers in 1) quickly evaluating code function and relevance, 2) enabling bi-direction exploration of code changes and goals, 3) clearly identify historical code that has been removed, and 4) leverage web searches to identify possibly relevant subgoals within the code history.

## REFERENCES


[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1199–1210. https://doi.org/10.1109/ICSE.2019.00122

[2] Omar Alghamdi, Sarah Clinch, Mohammad Alhamadi, and Caroline Jay. 2023. Novice Programmers Strategies for Online Resource Use and Their Impact on Source Code. In *2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*. 92–104. https://doi.org/10.1109/CHASE58964.2023.00018

[3] Karen Arnold. [n. d.]. *Ostrich Vintage Clipart.* https://www.publicdomainpictures.net/en/view-image.php?image=407552&picture=ostrich-vintage-clipart License: CC0 Public Domain.

[4] Rajiv D Banker, Robert J Kauffman, and Dani Zweig. 1993. Repository evaluation of software reuse. *IEEE Transactions on Software Engineering* 19, 4 (1993), 379–389.

[5] BH Barns and Terry B Bollinger. 1991. Making reuse cost-effective. *IEEE software* 8, 1 (1991), 13–24.

[6] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

[7] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. 2002. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering* 7 (2002), 115–156.

[8] Richard Catrambone and Keith J Holyoak. 1989. Overcoming contextual limitations on problem-solving transfer. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 15, 6 (1989), 1147.

[9] Catherine A Clement, D Midian Kurland, Ronald Mawby, and Roy D Pea. 1986. Analogical reasoning and computer programming. *Journal of Educational Computing Research* 2, 4 (1986), 473–486.

[10] Coder Coder. 2022. *Live coding a WORDLE clone (5 hrs) | HTML Sass JS.* YouTube. https://www.youtube.com/watch?v=PNGgQzw6PQg

[11] Cynthia L Corritore and Susan Wiedenbeck. 2001. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies* 54, 1 (2001), 1–23.

[12] Igor Crk and Timothy Kluthe. 2016. Assessing the contribution of the individual alpha frequency (IAF) in an EEG-based study of program comprehension. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 4601–4604.

[13] Françoise Détienne. 2007. Reasoning from a schema and from an analog in software code reuse. *arXiv preprint cs/0701200* (2007).

[14] K. Erdos and H.M. Sneed. 1998. Partial comprehension of complex programs (enough to perform maintenance). In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*. 98–105. https://doi.org/10.1109/WPC.1998.693322

[15] Andrew Forward and Timothy C Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*. 26–33.

[16] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Deutschland GmbH.

[17] David Garlan, Robert Allen, and John Ockerbloom. 1995. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th international conference on Software engineering*. 179–185.

[18] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2004. Analogical encoding: Facilitating knowledge transfer and integration. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 26.

[19] Mary L Gick and Keith J Holyoak. 1980. Analogical problem solving. *Cognitive psychology* 12, 3 (1980), 306–355.

[20] Paulo Gomes, Francisco C Pereira, Carlos Bento, and JL Ferriera. 2001. Using analogical reasoning to promote creativity in software reuse. In *Procs. of the Workshop Programme of the Fourth International Conference on Case-Based Reasoning*. 152–158.

[21] Neal Grandgenett and Ann Thompson. 1991. Effects of guided programming instruction on the transfer of analogical reasoning. *Journal of Educational Computing Research* 7, 3 (1991), 293–308.

[22] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code reuse in open source software. *Management science* 54, 1 (2008), 180–193.

[23] Mehdi T Harandi. 1993. The role of analogy in software reuse. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*. 40–47.

[24] Björn Hartmann, Scott Doorley, and Scott R Klemmer. 2008. Hacking, mashing, gluing: Understanding opportunistic design. *IEEE Pervasive Computing* 7, 3 (2008), 46–54.

[25] Ahmed E. Hassan. 2008. The road ahead for Mining Software Repositories. In *2008 Frontiers of Software Maintenance*. 48–57. https://doi.org/10.1109/FOSM.2008.4659248

[26] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A. Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristof Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matúš Sulír, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. 2022. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering* 27, 6 (July 2022), 125. https://doi.org/10.1007/s10664-021-10083-5

[27] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, San Francisco, CA, USA, 121–130.

[28] Keith J Holyoak, Paul Thagard, and Stuart Sutherland. 1995. Mental leaps: analogy in creative thought. *Nature* 373, 6515 (1995), 572–572.

[29] Ellis Horowitz and John B. Munson. 1984. An Expansive View of Reusable Software. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 477–487. https://doi.org/10.1109/TSE.1984.5010270

[30] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A Myers. 2022. Understanding How Programmers Can Use Annotations on Documentation. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 69, 16 pages. https://doi.org/10.1145/3491102.3502095

[31] Tomas Isakowitz and Robert J Kauffman. 1996. Supporting search for reusable software objects. *IEEE Transactions on Software engineering* 22, 6 (1996), 407–423.

[32] David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. In *2011 33rd International Conference on Software Engineering (ICSE)*. 351–360. https://doi.org/10.1145/1985793.1985842 ISSN: 1558-1225.

[33] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 344–353. https://doi.org/10.1109/ICSE.2007.45

[34] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[35] Charles W. Krueger. 1992. Software Reuse. *ACM Comput. Surv.* 24, 2 (jun 1992), 131–183. https://doi.org/10.1145/130844.130856

[36] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1937117.1937125

[37] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. https://doi.org/10.1145/1134285.1134355

[38] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and software* 7, 4 (1987), 325–339.

[39] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1987. Mental models and software maintenance. *Journal of Systems and Software* 7, 4 (1987), 341–355.

[40] Katsuhisa Maruyama, Takayuki Omori, and Shinpei Hayashi. 2016. Slicing Fine-Grained Code Change History. *IEICE Transactions on Information and Systems*

E99.D, 3 (2016), 671–687. https://doi.org/10.1587/transinf.2015EDP7282
[41] Travis Maynard. 2017. *Getting Started with Gulp–Second Edition*. Packt Publishing Ltd.
[42] H. Mili, F. Mili, and A. Mili. 1995. Reusing software: issues and research directions. *IEEE Transactions on Software Engineering* 21, 6 (1995), 528–562. https://doi.org/10.1109/32.391379
[43] Audris Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. 7–7. https://doi.org/10.1109/FLOSS.2007.10
[44] Israel J. Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. 2014. A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE Software* 31, 2 (2014), 78–86. https://doi.org/10.1109/MS.2013.142
[45] Jungkook Park, Yeong Hoon Park, Suin Kim, and Alice Oh. 2017. Eliph: Effective Visualization of Code History for Peer Assessment in Programming Education. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*. Association for Computing Machinery, New York, NY, USA, 458–467. https://doi.org/10.1145/2998181.2998285
[46] Nancy Pennington. 1987. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop, 1987*. 100–113.
[47] Brian H Ross. 1987. This is like that: The use of earlier problems and the separation of similarity effects. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 13, 4 (1987), 629.
[48] Geof Sheppard. [n. d.]. *Barnett Class ON939 rudder and starboard propeller*. https://link.to/original/image https://commons.wikimedia.org/wiki/File: Barnett_Class_ON939_rudder_and_starboard_propeller.jpg.
[49] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/1181775.1181779
[50] Manuel Sojer and Joachim Henkel. 2010. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems* 11, 12 (2010), 868–901.
[51] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on software engineering* 5 (1984), 595–609.
[52] George Spanoudakis and Panos Constantopoulos. 1994. Similarity for analogical software reuse: A computational model. In *ECAI*. PITMAN, 18–22.
[53] Alistair Sutcliffe and Neil Maiden. 1991. Analogical software reuse: Empirical investigations of analogy-based reuse and software engineering practices. *Acta Psychologica* 78, 1-3 (1991), 173–197.
[54] Karen Swan. 1991. Programming objects to think with: Logo and the teaching and learning of problem solving. *Journal of Educational Computing Research* 7, 1 (1991), 89–112.
[55] Antero Taivalsaari, Tommi Mikkonen, and Niko Mäkitalo. 2019. Programming the Tip of the Iceberg: Software Reuse in the 21st Century. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 108–112. https://doi.org/10.1109/SEAA.2019.00025
[56] Unknown. [n. d.]. *Plane Image*. https://creativecommons.org/publicdomain/zero/1.0/ License: CC0 1.0.
[57] Paul Wuilmart, Emma Söderberg, and Martin Höst. 2023. Programmer Stories, Stories for Programmers: Exploring Storytelling in Software Development. In *The 9th Edition of the Programming Experience Workshop*.
[58] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (Oct. 2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091 Conference Name: IEEE Transactions on Software Engineering.
[59] YoungSeok Yoon and Brad A. Myers. 2015. Semantic zooming of code change history. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 95–99. https://doi.org/10.1109/VLHCC.2015.7357203
[60] YoungSeok Yoon and Brad A. Myers. 2015. Supporting Selective Undo in a Code Editor. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 223–233. https://doi.org/10.1109/ICSE.2015.43 ISSN: 1558-1225.