



A Qualitative Study on the Implementation Design Decisions of Developers

Jenny T. Liang
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
jtliang@cs.cmu.edu

Maryam Arab
Department of Computer Science
George Mason University
Fairfax, VA, USA
marab@gmu.edu

Minhyuk Ko
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
minhyukko@vt.edu

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

Thomas D. LaToza
Department of Computer Science
George Mason University
Fairfax, VA, USA
tlatoya@gmu.edu

Abstract—Decision-making is a key software engineering skill. Developers constantly make choices throughout the software development process, from requirements to implementation. While prior work has studied developer decision-making, the choices made while choosing what solution to write in code remain understudied. In this mixed-methods study, we examine the phenomenon where developers select one specific way to implement a behavior in code, given many potential alternatives. We call these decisions *implementation design decisions*. Our mixed-methods study includes 46 survey responses and 14 semi-structured interviews with professional developers about their decision types, considerations, processes, and expertise for implementation design decisions. We find that implementation design decisions, rather than being a natural outcome from higher levels of design, require constant monitoring of higher level design choices, such as requirements and architecture. We also show that developers have a consistent general structure to their implementation decision-making process, but no single process is exactly the same. We discuss the implications of our findings on research, education, and practice, including insights on teaching developers how to make implementation design decisions.

Index Terms—implementation design decisions, software design

I. INTRODUCTION

Making decisions effectively is a crucial skill for software engineers [1]. One reason is because making explicit and rationalized design decisions during the design process improves software design quality [2]. Developers make these explicit decisions throughout the software design process, from requirements and architecture to implementation. These decisions in turn have downstream effects on the software, such as influencing how easily developers comprehend a codebase [3] or resulting in systems that are difficult to maintain [4], [5].

At higher levels of software design, developers make explicit decisions about the software architecture by prototyping them at the whiteboard [6] or documenting them in UML diagrams [7]. Researchers have developed an understanding of how developers make architectural decisions [8], [9], even

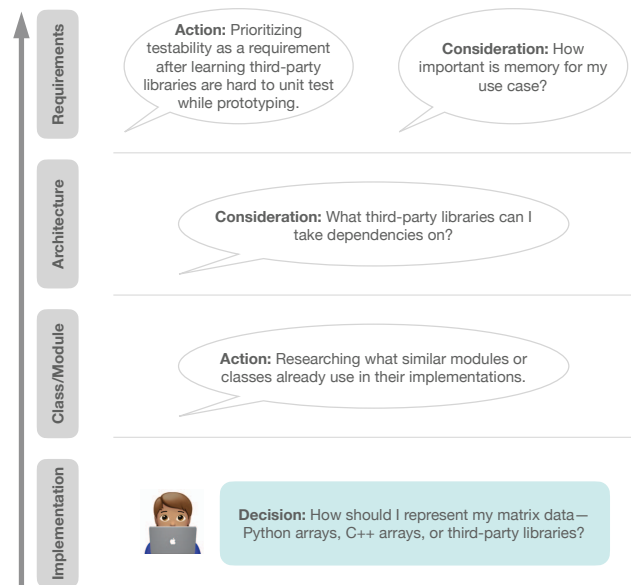


Fig. 1. An example of an implementation design decision. Developers consider aspects of software design that are above the implementation (e.g., requirements, architecture, or class and modules) to make these decisions.

building several tools to aid this process [10], [11]. At the code level, developers also make explicit decisions, such as implementing specific design patterns [12], [13]. Yet, the decisions that developers make while choosing what solution to write in code remains understudied. We call these decisions *implementation design decisions*.

Implementation design decisions are when developers select one specific way to implement a behavior, given many potential alternatives. They may choose an implementation that minimizes the time to market in place of producing high-quality, robust software. Meanwhile, developers may choose

to implement solutions for other reasons: system requirements (e.g., performance), code quality (e.g., readability), or convenience (e.g., ease of implementation).

Figure 1 includes an example of an implementation design decision. A developer is considering three different ways to represent their matrix data in a script—Python arrays, C++ arrays, or using a third-party library. If the developer wanted to minimize runtime, they could use C++ arrays or third-party libraries. If they wanted a simple, easy-to-read solution for their teammates, they could opt for Python arrays.

Understanding implementation design decisions can provide insights on what decisions result in good or bad software designs, which could be taught to novice developers. Furthermore, studying these decisions can elicit a broader set of considerations developers are optimizing for. This could help explain the decisions that are on its face sub-optimal but are in fact necessary (e.g., decisions causing technical debt).

In this work, we study implementation design decisions with a focus on the explicit decisions made by developers, as conscious reasoning in software design improves the design quality [2]. We address the following questions:

- **RQ1:** What implementation design decisions do software developers make?
- **RQ2:** What considerations do software developers have while making implementation design decisions?
- **RQ3:** What process do software developers follow to make implementation design decisions?
- **RQ4:** Which types of developer expertise are described in the implementation decision-making process?

To answer these questions, we designed a mixed-methods study using surveys and interviews to understand implementation design decisions. Our study had 53 participants who program professionally. We find that implementation design decisions demanded careful thought. Overall, they require constant evaluation of higher levels of design and could even exert influence on them (see Figure 1), which corroborates findings from prior work [14]. This is in contrast to other work that characterizes implementation as a natural result from higher level forms of design in a top-down fashion. For example, Perry and Wolf described implementation as a code-level representation satisfying requirements, architecture and design [15]. Thus, our study supports the theory that problems and solutions in software design co-evolve with one another—as the solution develops, the problem space can update [8]. We also show that developers have a general structure to their decision-making process, but each developers' process is unique. Finally, we provide implications in how our results can be applied in research, education, and practice.

II. RELATED WORK

A. Implementation Design Decisions

Ralph and Tempero studied the characteristics of decisions made while programming [14]. They elicited common themes in these decisions, such as their considerations and the approaches taken to solve a problem. Our study examines a

similar phenomenon as Ralph and Tempero. It extends this work by also investigating the systematic processes developers use to make such decisions, as well as the types of expertise they apply in making them.

Prior research has also investigated a specific outcome from a particular subset of implementation design decisions in depth—technical debt. These occur when developers choose implementations that bias time to market over software quality [16]. Technical debt comes in several forms, such as architectural, structural, documentation, test, infrastructure, and requirements debts [4], [5]. Technical debt has many negative downstream effects, such as reducing developer productivity [17] as well as decreasing team morale, causing delays, and lowering code quality [18]. Technical debt can be caused for many reasons, including planning and management (e.g., deadlines), development issues (e.g., not adopting good practices), software engineering processes (e.g., a lack of documentation), a lack of knowledge (e.g., lack of experience), human factors (e.g., lack of commitment), organizational factors (e.g., lack of trained professionals), external factors (e.g., pressure), and infrastructure (e.g., unavailable infrastructure) [5], [19].

Implementation design decisions, such as ones that result in technical debt, are concerned with decisions that developers make when they are about to write an implementation. They consider different ways a behavior may be implemented in code. Thus, prior work in this space largely focuses on the far-reaching effects of implementation design decisions, rather than the decisions themselves.

B. Decision-Making in Software Engineering

Developers make decisions across several types of activities, such as in project planning and verification [20]. These decisions require developers to make tradeoffs by reasoning about future outcomes [21]. Prior work in software engineering expertise also suggests that knowing how to effectively make decisions is a form of expertise [1], [22], [23].

Requirements engineering activities can be framed as decision-making processes at the organizational and individual levels [24]. While deciding on requirements, software developers make decisions on which requirements to prioritize [25].

In software architecture, developers make decisions to select between candidate architectures [26], [27] and integrate components into an existing system [20]. van Vliet et al. argue that software architecture can be viewed as a set of design decisions made by software architects [8]. Additionally, software architecture decisions use both slow, rational and fast, intuitive thinking and can be prone to cognitive biases [8], [9]. Zannier et al. found which of the two thinking approaches designers used depended on how structured the problem was [28].

Software developers also make decisions in API design. Stylos and Myers outlined the space of design decisions for APIs, such as design patterns to use or what fields or methods to provide. These decisions were split across the architectural, structural, class, and language levels [29]. In order to make proper API design decisions, software practitioners and orga-

nizations have outlined several guidelines [30], [31] and online resources [32] on how to design APIs.

Overall, prior work has largely studied developers' decision-making in specific development contexts or at higher levels of design, such as requirements or architecture. Our work diverges from this literature by studying the decisions made while developers are about to write implementations in code.

C. Software Design Practices

Many studies have examined how software developers do software design. Petre and van der Hoek studied the individual practices of software designers [33], such as involving experts from outside the team to learn domain-specific knowledge. Practitioners have also written resources on writing code with clean software design (e.g., [34], [35]) and architecture (e.g., [36], [37]) with prescriptions on how to make design decisions effectively, such as following well-known programming principles like SOLID. In open-source software, contributors make design decisions on bug reports [38] and online discussions on GitHub issues [39], [40].

For software design at higher levels of abstraction, developers are often known to perform design activities at the whiteboard, creating visual diagrams to support the design process [6], [41]. Software developers can informally denote their designs using UML [7] or sketches [42]. Prior work has also investigated the process software engineers follow while doing early stage design work. Sharif et al. found that while designing, developers engage in activities which overlap with the traditional software development process, namely requirements, analysis, design, and implementation [43].

Thus, prior research has investigated the practices developers follow to design software. Instead of studying developers' actions, our work contrasts prior literature by studying the decision-making and reasoning of developers.

III. STUDY DESIGN

To answer the research questions, we collected data on software developers' implementation design decisions from 53 study participants (Section III-A) using surveys (Section III-B) and interviews (Section III-C). We then analyzed the data using qualitative coding (Section III-D). The breadth of the survey data and the depth of the interview data corroborated our findings from multiple sources.

A. Participants

To elicit a wide range of developers' insights on implementation design decisions, we recruited developers with diverse programming experiences. Our inclusion criteria was developers who contributed their programming expertise to at least one project in a professional setting.

1) *Sampling strategy*: We recruited participants with different levels of software engineering experience, technology expertise, job titles, and engineering team sizes. The authors then released a survey (Section III-B) on their personal social media accounts for recruitment. Social media posts displayed

brief descriptions about implementation design decisions, the estimated time of completion, and a survey link.

We increased sample coverage by also recruiting on Reddit. The first author advertised the survey in a text post on 10 popular developer-centric Reddit communities. As of January 2022, the subreddits had between 6,681 members to 291,170 members. Reddit communities were selected by technology (e.g., *r/reactjs*, *r/php*), geographic location (e.g., *r/developersIndia*), or role (e.g., *r/dataengineering*). Posts were only published after receiving permission from the community's moderators, as stipulated by the institutional review board. The Reddit recruitment posts introduced implementation design decisions, provided an estimated time of completion, explained why the subreddit was a good fit, and linked the study. We also relied on snowball sampling by encouraging interview participants to share the study to others. In the survey, participants indicated if they wanted to participate in a follow-up interview.

In total, 60 developers agreed to participate in the study, with 53 who met the inclusion criteria and participated in the study. 46 participants completed the survey and 14 participants completed an interview.

2) *Demographics*: We report interviewee demographics in Table I. In our study, participants represented diverse geographic locations, including the Americas ($n = 27$), Africa ($n = 1$), Asia ($n = 7$), Europe ($n = 14$), and Oceania ($n = 3$). Multiple genders were also represented, such as man ($n = 44$), woman ($n = 7$), and non-binary ($n = 1$). Participants had job titles such as junior, senior, or principal software engineer; Chief Technology Officer; software architect; machine learning engineer; and research engineer. Participants reported contributing between 1 to over 1,000 projects, with a median of 25. Participants worked in companies of varying sizes, whose engineering organization sizes ranged from 1 to 36,000, with a median of 18. Participants reported using a variety of technologies (e.g., Java, MongoDB, Angular, R, Verilog, Jupyter notebooks) and working on diverse problem domains (e.g., deployment infrastructure, IoT, static analysis tools, financial technology, healthcare, online exams).

B. Survey

We designed a 15-minute Google Forms survey on examples of implementation design decisions and considerations and distributed it to developers using the sampling strategy from Section III-A.

1) *Design*: In the survey, we first presented participants with the definition and three examples of implementation design decisions. We then asked participants to provide examples of implementation design decisions and considerations, limiting examples to the past five days in order to reduce memory bias. These questions had a response length minimum of 30 characters to encourage participants to record sufficiently detailed answers. Following best practices, we used the HCI Guidelines for Gender Equity and Inclusivity [44] to collect gender information. We allowed participants to select multiple responses for questions on gender. The full survey instrument is available in our supplemental materials [45].

TABLE I
OVERVIEW OF INTERVIEW PARTICIPANTS. “SOFTWARE ENGINEER” IS
ABBREVIATED AS “SWE”.

ID	Job title	# projects	Org. size	Gender	Location
P4	Senior Principal Engineer	50+	1,700	Man	United Kingdom
P6	Chief Technology Officer	30	16	Man	Germany
P7	Scientific SWE II	20	200	Woman	United States
P15	Head of Research	Dozens	12	Man	Israel
P18	Principal SWE	50	1,000	Man	United States
P22	Chief Technology Officer	10	6	Man	Philippines
P28	Data Platform Engineer	5-6	500	Man	Australia
P29	Data Engineer	15-20	12	Woman	Germany
P31	Senior SWE / Analyst	200+	9	Man	United States
P32	SWE	4	20,000	Woman	United States
P34	Senior SWE	20	50	Man	United States
P50	Technical Director; Senior SWE	25+	10	Man	United Kingdom
P51	Small Business Owner	100's	2	Man	Canada
P52	SWE	50	2	Man	United States

2) *Piloting*: Following best practices for experiments with human subjects in software engineering [46], we conducted pilots of the survey to identify and reduce confounding factors. We piloted drafts of the survey with four software developers to clarify wording and updated the survey after each round of feedback. To ensure data quality, the survey was deployed publicly, piloted, and updated on the first 15 survey responses.

C. Interviews

We conducted 45-minute interviews via online conference calls to gather examples of implementation design decisions and considerations as well as developers’ decision-making process. Interviews were recorded and transcribed. Recordings were destroyed after transcription. Interview participants were compensated with a \$30 USD Amazon.com gift certificate.

1) *Design*: Topics in the interview included implementation design decisions and considerations participants made in the past five days, as well as written explicit programming strategies on how participants made their decisions. We collected written explicit programming strategies as a structured format to capture developers’ processes and knowledge. An example of an explicit programming strategy from our study is in Figure 2; we report all collected strategies in our supplemental materials [45].

Explicit programming strategies are “human-executable procedure[s] for accomplishing a programming task” [47]. In this study, explicit programming strategies represent the process software developers used to make implementation design decisions. We collected them since developers can follow systematic processes to make some design decisions, such as selecting between multiple software architecture alternatives [26], [27]. Additionally, developer expertise and processes can also be externalized by developers via explicit programming strategies [48], which allowed us to elicit software developers’ decision-making processes in interviews.

2) *Protocol*: Two authors conducted the interviews: one to execute the protocol and one to record the participant’s explicit programming strategy. Having a interviewer experienced in strategy authoring to write strategies ensured strategy writing quality, as authored strategies could be ambiguous or struggle to generalize [48]. During the interview, we reminded participants of the definition of implementation design decisions in a Google Slides presentation, using similar wording and examples from the survey. Next, we collected implementation design decisions and considerations. Finally, for as many design decisions that time allowed for, we extracted an explicit programming strategy the participant used to make their decision. The interview protocol is available in our supplemental materials [45].

To extract programming strategies, the participant explained their decision-making process step-by-step. Interviewers translated this to an explicit programming strategy in a shared Google Document to reduce the participant’s cognitive load while recalling their process, as strategy authoring is cognitively demanding [48]. The participant then reviewed the strategy and provided corrections or feedback. Because authored strategies may omit details that prevents the strategy from being usable [48], the participants elaborated on edge cases and updated their strategy accordingly to increase its robustness. Since authored strategies’ scope may be too narrow [48], interviewees updated the strategy to be general enough for a similar problem. After reviewing the strategy, the participant edited wording and clarity so the strategy was at a quality that could be released publicly. Following prior work [49], we documented a brief description of the strategy; tools, technologies, and prior knowledge necessary to use the strategy; and the steps of the strategy. The authors updated the strategy for consistency, but participants could edit the strategy upon request. Participants also could add additional data (e.g., comments, visual media), to explain their process.

3) *Piloting*: Following best practices in experiments with human subjects in software engineering [46], we piloted the interview to identify and reduce confounding factors. The first author ran the interview protocol on one author and three developers and updated the protocol based on their feedback. The purpose of the pilots was two-fold: 1) to improve the clarity of the interview protocol and 2) validate the hypothesis that developers had systematic processes to make implementation design decisions and further, could articulate their processes. We found that all pilot participants could recall and articulate

strategies for making implementation design decisions.

Use this when: *Using less common features in libraries instead of using the popular functions*

Tools/technologies: *StackOverflow, Google, continuous learning*

Prior knowledge: *Common design patterns, popular libraries*

- 1) *Decide what the goal of the program is.*
- 2) *Begin writing the program.*
- 3) *While writing the program, search online whether other libraries support your use case. Use your prior knowledge or colleagues to help guide your search.*
- 4) *Choose a library which meets your use case. This can be based on the popularity of the library with respect to the language.*
- 5) *Look at the features of the library and test the ones that you're interested in on small examples. Get a feel of the library and select a solution which achieves the desired behavior.*
- 6) *If you have code that works, show the solution to another individual for review.*

Fig. 2. An example of an explicit programming strategy from the study. User-generated content is written in *italics*.

D. Analysis

To analyze the collected data, we used qualitative coding. We *open coded* the data for **RQ1**, **RQ2**, and **RQ3** to summarize the data on various aspects of implementation design decisions as this phenomenon is understudied. We *close coded* **RQ4** to situate the strategies with prior work on software engineering expertise, which has been well-studied (e.g., [1], [22], [23], [50], [51]).

1) *Open Coding:* For *open coding*, we followed best practices by Hammer and Berland [52], which outlines procedures on interpreting coding results and reporting coding disagreements. We treated generated codes as tabulated claims about the data that could be investigated in future work. We checked the reliability of the coding by resolving disagreements, first by discussing any disagreements and then coming to an agreement as a group. Finally, we interpreted coding disagreements as coding variance and reported the content of the disagreements.

We followed best open coding practices [53], preparing separate documents for each coder for qualitative analysis, taking care to remove the prompts from the responses. In these documents, survey responses and interview responses were stored separately and analyzed independently, as the data was collected in different contexts. We also shuffled all the rows in the data to remove any ordering effects. Finally, we removed data collected from piloting.

Open coding occurred in multiple phases. In the first phase, three authors separately reviewed the responses and inductively generated codes for each dataset. Each response was labeled with zero or more codes. Each code was given a unique identifier and a brief description. To aggregate the codes, the authors compared their separately generated codes and identified codes with similar concepts. These codes were merged under a single code and copied to a shared codebook. For the remaining codes, the authors discussed instances of

disagreement and resolved them by unanimously agreeing to add or remove the code in the shared codebook. Disagreements were most frequently the result of differing scopes of codes, rather than the meaning of the participants' statements. Some disagreements arose due to an author not coding a part of the response another author did. In the second round, the authors applied the shared codebook to the original data. If there were multiple datasets to analyze for the same research question, each dataset was coded using the aforementioned process. Then, the resulting codebooks were merged by the authors. The authors identified codes with similar definitions and added them to a final codebook with a unique identifier. The remaining unmerged codes were automatically added to the final codebook. The authors performed a third round of coding with the final merged codebook. For **RQ3**, the authors then applied pattern coding to the final codes to group the codes into broader categories [54]. To do this, the authors placed each code into an initial category by unanimously agreeing to put it in a category or create a new one. Then, they reviewed each category and unanimously finalized its scope and, if necessary, moved the codes between categories to reflect the new scope.

2) *Closed Coding:* For *closed coding*, the first author identified a codebook to code the dataset with. For **RQ4**, we used Table 3 and Table 5 from Li et al. [1], which respectively contains codes on attributes of expert developers' decision-making processes and their software and designs. The three authors deductively applied the codes to the dataset. Each instance was labeled with zero or more codes. Next, they reconvened to discuss instances of disagreement and resolved them by unanimously agreeing on which codes were to be applied. In this step, disagreements arose due to the scope of the code rather than the meaning of the statements.

3) *Data:* For **RQ1**, we analyzed 82 examples of implementation design decisions from the survey and interviews. For **RQ2**, we analyzed 113 examples of considerations from survey and interview data as well as implementation design decisions from interviews, since participants mentioned considerations in context of their decisions. For **RQ3** and **RQ4**, we analyzed a dataset with 99 steps from the 16 collected strategies.

In addition to extracting action codes for **RQ3**, we analyzed the programming strategies on the decision-making process as a sequence of action codes and categories. We did this by replacing each step of the strategy with its respective code or category. If a step had multiple codes or categories, we represented the step as a sequence of codes or categories in the order they were mentioned. If a code or category occurred consecutively, they were reduced to a single occurrence. We include these representations of strategies in the supplemental materials [45].

IV. RESULTS

We report the results to our research questions below. Due to space constraints, we only discuss codes we found interesting with respect to prior work.

TABLE II
THE TYPES OF IMPLEMENTATION DECISIONS MADE BY SOFTWARE DEVELOPERS. CODES DISCUSSED IN DETAIL ARE UNDERLINED.

Code & Description	Representative Quote
<i>Alternatives</i> —Deciding what high-level approaches to use to address a particular problem.	“So we are ingesting data. One way...is using Python scripts... The other option that we looked into was getting it via third party tools...the third was just outsourcing it.” (P28)
<i>Behaviors</i> —Deciding the program specification: what parameters to set for a program and their types, what outputs the program should give, and the behavior of the program.	“[The API] would expect to take in the input data type, which is this union of Xarray, Numpy, Dask, all of these supported data types...” (P7)
<i>Data</i> —Deciding how to manage data within software: what data should be handled in a program, how it should be represented, and how it should be interacted with.	“I opted to represent the tree as an ancestry string of the top slash the next, [and] the next. And then you can use ‘like’ with a wild card and you’ll get the subtree.” (P31)
<i>Code constructs</i> —Deciding which programming language constructs to use within a program.	“Making a change to a Python program, I removed an indexing expression (<code>val = x[0]</code>) and replaced it with a destructuring assignment (<code>val, _ = x</code>).” (P18)
<i>Structure</i> —Deciding how to organize the codebase, where files should lie, and how code should be modularized.	“I’m going to refactor this to bring out the bits of logic that pertain to...my bit of the business, so that I can then later have ownership of it...rather than [having a] big monolithic system.” (P4)
<i>Programming languages, APIs, services</i> —Deciding the programming languages, APIs, or third-party services to use in the software system or script.	“I used Golang to handle a large amount of JSON files that would’ve taken too long to handle in Python.” (P20)
<i>Automation</i> —Deciding whether to implement a technology solution from scratch.	“I could have manually typed in the new kinds of records that I wanted in production...but instead I encoded that all in a formalized runnable script.” (P32)
<i>Reuse</i> —Deciding whether code should be reused and to what extent it should be general enough to be extended to different scenarios.	“Merge two C# applications (FTP and SFTP server) into one, in order to reuse file tree state, user authentication, and so on.” (P45)
<i>Updates</i> —Deciding whether to update the software or not.	“Do I tell them I can’t fix the problem or do I go in and tempt small solutions, just to get it to function for a few days...or do I completely write my own fix?” (P51)

Prior work (e.g., [15], [55]) characterizes implementation as naturally arising from higher levels of design. In contrast, we found that implementation design decisions involved a constant top-down and bottom-up dialogue between implementation and higher levels of design, such as requirements and architecture. This supports the view that problems and software implementations co-evolve with one another [8]. Consider Figure 1 as an example. When a developer decides whether their matrix-based data should be represented using native Python arrays, C++ arrays, or a third party library (e.g., Numpy), they may consider the architecture by thinking about what languages or third-party libraries are compatible with their system. They may also consider non-functional requirements (e.g., memory or performance) or look to how other similar modules address this problem. Finally, they may even change the priority of certain requirements, such as testability, after realizing the difficulty of prototyping unit tests with a third party library.

A. What implementation design decisions do software developers make? (RQ1)

Participants described 8 different types of implementation design decisions (see Table II). Each code appeared more than once in our data. All codes appeared in all datasets.

a) *Behaviors*: Participants decided on how the software should behave, such as its inputs, outputs, and what should occur during execution. This was an informal version of the requirements elicitation, analysis, and validation processes [56], [57]. These decisions often required a change in requirements,

which corroborates the viewpoint that decisions about the solution may change requirements [8]. For instance, participants decided on entire method specifications when requirements were under-specified and made changes to the program behavior to handle certain requirements:

“*But, I decided to record each line of the CSV file as a record with a header record...So on that header record, I record who gave me the file and when it was given to me. I could reproduce the CSV file from what I’m storing.*” (P31)

b) *Code constructs*: Consistent with prior work [14], participants debated which programming constructs to use (e.g., loop constructs, ternary operators, pointers). These decisions were the lowest level decisions made. Even at this level, participants still considered requirements (e.g., performance):

“*[Sometimes] I would rather go for efficiency or performance [but when]...I am working with other developers, I’m more leaning to having the code more readable... Instead of functional programming mapreduce I go for loop, so that the other developers can understand the code itself.*” (P22)

c) *Updates*: Participants deliberated whether to make specific code changes (e.g., fixing a defect), as it could have unwanted effects. They modeled potential outcomes—a decision-making attribute of developers [1]. These decisions at times required considering the software architecture, such as while deciding whether to update dependencies:

“*The library that we use...didn’t compile anymore... We have two [options]. One is to say, ‘Okay, we freeze the library version...’ and then we postpone the solution of the problem. Or we look at the problem and fix it immediately.*” (P6)

TABLE III
SOFTWARE DEVELOPERS' CONSIDERATIONS FOR IMPLEMENTATION DECISIONS. CODES DISCUSSED IN DETAIL ARE UNDERLINED.

Code & Description	Representative Quote
<u>Community Support</u> —How well-supported by a developer community a technology is.	“I wanted to use PHP 8.1, but there is still no general support...” (P54)
<u>Features</u> —The features a technology contains.	“Open source C# MailKit was selected over builtin Smtplib to...allow flexible email body manipulation.” (P37)
<u>Popularity</u> —The number of users that use the technology or library.	“[Laravel SPatie Media Library] being very well understood by the rest of the Laravel developer community.” (P52)
<u>Reliability</u> —How reliable and correct the software is.	“Correctness [with] concurrent updates and...mutable objects.” (P44)
<u>Security</u> —How secure software is; robustness of software to adversarial attacks.	“Each decision had tradeoffs...[in] security (the latter being exposing, possibly private, brands.)” (P24)
<u>Maintainability</u> —How easily maintenance actions (e.g., fixing defects, updating components) can be performed on software.	“Over-engineering a system that may...add cognitive overhead to any maintenance.” (P39)
<u>Testability</u> —How easily software can be tested (e.g., unit tests).	“Testability (functional is almost always easier to test).” (P27)
<u>Extensibility</u> —How easily the code can be extended to accommodate changes (e.g., new features).	“So how the structure and application itself is laid out so that it’s not going to be a pain to work with, as we expand it.” (P50)
<u>Performance</u> —Performance aspects of the code (e.g., runtime, memory).	“The function call is expensive in certain...circumstances.” (P18)
<u>Reproducibility</u> —Whether code is able to reproduce the same output, given the same input.	“Does the code do it in an idempotent way? So...the same input would do the same output regardless of how many times you do it.” (P22)
<u>Requirements</u> —The requirements of the software; customer needs.	“After the first implementation, a new requirement came so structuring for reuse [was] useful.” (P15)
<u>Future Requirements</u> —Requirements or customer needs that may or may not occur in the future.	“Will there be a need to run the pipeline in parallel some day (like on Spark or Dask)?” (P27)
<u>Skills</u> —The current skills of the team or of the developer.	“We also chose PHP because...more developers familiar with the PHP framework than with Python frameworks.” (P56)
<u>Budget</u> —Amount of resources (e.g., time, money) available to implement the software project.	“Because we were on a tight deadline...I decided to just process [the data] all on my local machine...and then upload it.” (P57)
<u>Reusing Resources</u> —Reusing existing resources (e.g., code, practices).	“What parts of the code will they reuse?” (P32)
<u>Difficulty</u> —How much effort completing the implementation will be.	“The implementation difficulty comes into [these decisions].” (P28)
<u>Readability</u> —How easily code syntax is read by a developer.	“Generics in code may be harder to comprehend for some...”(P42)
<u>Code Cleanliness</u> —The quality of the implementation’s code; how easy it is to onboard other developers and make updates.	“Try not to make ravioli code where we have too many modules that do little things.” (P42)
<u>Simplicity</u> —The length or complexity of the implementation.	“I did this to keep my pull request shorter and closer to the original code. Less to read means faster code review.” (P18)
<u>Consistency</u> —Being consistent with the code style of the programming language or code base.	“So, not only is it existing code that’s already there. I don’t want to be the person to introduce something different.” (P34)
<u>System Fit</u> —How well the implementation fits in with an existing code base or system.	“Where to set up the event subscription... [In] a React component or outside of the React/Redux content...” (P24)
<u>Data</u> —How data in the system will be managed or handled.	“This is a trade-off of having less-fresh data, with being more robust in the event the 3rd party is unavailable.” (P31)
<u>Impacts</u> —The impacts that the implementation may cause.	“I want to be very safe when making potentially-impactful changes in production environments.” (P59)
<u>Users</u> —Thinking about collaborators who will be working in the code base; the usability of the software for end-users.	“It’s trying to make the code...understandable for the other developers for maintenance purposes and if they need to upgrade the code...” (P22)
<u>Documentation</u> —Writing documentation for the implementation.	“So that’s a significant documentation...cost.” (P50)

B. What considerations do software developers have while making implementation design decisions? (RQ2)

Participants described 25 distinct considerations while making implementation design decisions. We report them in Table III. Each code appeared more than once—17 codes appeared in all datasets, 7 codes appeared in two datasets, and 1 code appeared in one dataset.

a) *Community support*: Participants reported that community support for third-party libraries was a factor. This

ensured that dependencies were well-maintained for software quality. Having community support enables the production of educational resources for the tool, such as on YouTube [58] as well as StackOverflow and blog posts [59]. Participants said this reduced onboarding costs:

“And there are instructional videos on YouTube and whatnot [that] can already teach people how to do [use the tool] without the rest of the development team having to do anything.” (P52)

b) *Future requirements*: Similar to the management phase in requirements engineering [56], [57] participants noted

requirements could change. Understanding future requirements ensured the software was useful in the long term. Estimating them depended on prior experience and domain knowledge:

“...I’m making an assumption about what might come down the pipe in the future. [It] depends on kind of my experience in that field, and my work that I’ve done with past clients to think that my future clients might be similar enough to them.” (P51)

c) *Consistency*: Similar to prior work [14], consistency of the code style in the codebase or following programming language convention was a consideration. This reduced confusion between teammates and cognitive overhead for individuals working across multiple contexts. This occurred both at the application- and module-levels:

“I have 200 or so web applications and having consistency...makes it easy for me to switch between them without having to re-remember a whole different framework.” (P31)

“So some places [a stock] is called a stock, some places it’s called security...I will try to keep that pattern, even if it’s something I don’t necessarily agree with.” (P34)

d) *System fit*: Participants considered how easily the implementation could be integrated with existing code, such as synergy with specific technologies. This is an attribute of expert developers’ software and designs [1]. System fit required knowledge of the system’s architecture:

“Choosing between django-q and celery was difficult- one is closely coupled with django environment and the other has long history/reliability.” (P55)

C. What process do software developers follow to make implementation design decisions? (RQ3)

We describe the types of actions developers take while making implementation design decisions (Sections IV-C1). We then report the sequence of actions that developers follow in their decision-making process (Section IV-C2).

1) *Actions*: Participants described 15 types of actions in their strategies. We grouped these into 7 categories: defining the problem space; ideating, evaluating, prototyping, implementing, and verifying potential solutions; and updating knowledge. Many of these actions overlapped with the traditional software development process, similar to prior work [43]. Furthermore, participants’ actions were often a dialogue between the implementation and requirements. The full list of the actions in implementation design decisions are in Table IV. All codes occur in the data at least twice.

a) *Updating requirements*: Study participants described times when requirements changed after they were defined. This occurred after learning from proof-of-concepts, reacting to changes in the situation, or analyzing the requirements’ viability. Unlike in requirements engineering [56], [57], requirements also changed during implementation. In these cases, updating requirements was how participants dealt with unforeseen circumstances during implementation, such as time constraints. Participants even developed heuristics to do so:

“If you are under time constraints, restrict the scope of your implementation and don’t let perfect be the enemy of good.” (P50)

b) *Evaluating*: Participants evaluated potential alternatives for pros and cons, where they compared them against a

list of considerations, especially non-functional requirements. Some participants wrote lists or drew matrices, while others developed checklists from their expertise:

“Decide what you think is a pro/con of a given solution for your use case.

- Security concerns, an insecure package is never acceptable...
- The popularity of the package is critical for evaluating lifetime reliability and long term support.
- Level of skill required to use the package, poorly designed apis will be difficult to extend if needs change, and complicated for junior developers to work with.
- Clean, consistent and clear are the ideal interfaces.
- Consider the cost to replace the package if licensing, support or project direction dramatically change...” (P50)

c) *Proof-of-concept*: Study participants reported creating proof-of-concepts, which is also important in requirements elicitation [56], [57]. This quickly determined whether a potential solution was viable. Participants varied in which solutions they chose to prototype—some chose to prototype only the best candidate solution, while others prototyped all potential solutions. Prototyping was also an information gathering mechanism to brainstorm potential solutions:

“Look at the features of the library and test the ones that you’re interested in on small examples. Get a feel of the library...” (P6)

Participants developed a proof-of-concept to update requirements. This was one way they considered a higher level of design during implementation design decisions:

“Ask the people who you interviewed to try your function and provide feedback on any of the parameters.” (P7)

d) *Researching*: Participants researched the problem space, as identified in prior work [14]. This was the second most reported action. This action was often used to elicit requirements. Participants worked with stakeholders (e.g., project managers) and accessed websites for knowledge sharing in software engineering, such as StackOverflow [59] and Reddit [60]. They reviewed similar projects and used empirical methods to understand the problem and generate requirements:

“[Using] tools to go through Git history...to find potential problems...” (P4)

2) *Processes*: Just as how software designers follow individualized processes [43], we found that strategies about developers’ decision-making processes were unique: there were no repeated sequences of action codes or categories in the strategies. One source of dissimilarity were action codes and categories that were repeated in strategies. Table IV shows the occurrences of repeated action categories in the strategies. All action categories were repeated except ideating actions. Implementing and defining problem space actions were most repeated in participants’ strategies. Yet, there were commonalities in the structure, namely when certain types of actions occur. This is shown in Table V, which reports the median position of each action category across all action sequences.

D. Which types of developer expertise are described in the implementation design decision-making process? (RQ4)

Table VI shows the types of developer expertise in study participants’ decision-making process. We found that decision-making expertise was most commonly cited in strategies.

TABLE IV
SOFTWARE DEVELOPERS' ACTIONS IN MAKING IMPLEMENTATION DECISIONS. CODES DISCUSSED IN DETAIL ARE UNDERLINED. THE NUMBER OF TIMES AN ACTION CATEGORY IS REPEATED WITHIN THE SAME STRATEGY IN OUR DATA IS DENOTED WITH ×.

Code & Description	Representative Quote
Defining Problem Space (×8)	
<i>Providing Context</i> —Explaining context about the problem the developer is facing (e.g., refactoring).	“Write an initial program...If you have another program that requires a similar behavior, consider whether you want to refactor the code.” (P15)
<i>Defining Requirements</i> —Defining the requirements of the solution, considering user needs, business needs, and organization needs.	“...Figure out what use cases [your end users] would want for this function. Ask your end users to provide examples of inputs...” (P7)
<i>Updating Requirements</i> —Updating the requirements of the solution after they are initially defined.	“If you find new requirements from your proof-of-concept, go to step 1.” (P52)
Ideating (×0)	
<i>Brainstorming</i> —Brainstorming potential solutions that could solve the problem.	“Think about the problem for a set period of time and write down more than one idea on how to implement a solution...” (P7)
Assessing (×4)	
<i>Evaluating</i> —Evaluating the developer’s current situation; considering the pros and cons for each solution.	“List out all the options that you have into a document and their pros and cons.” (P29)
<i>Estimating</i> —Estimating the potential costs associated with implementation.	“Determine whether the requirements are realistic given the resources you have available.” (P34)
Prototyping (×1)	
<i>Proof-of-Concept</i> —Building a proof-of-concept for a potential solution.	“Test each option in the development environment...” (P29)
Implementing (×15)	
<i>Choosing</i> —Choosing a solution to implement.	“Select the option which meets your requirements...” (P18)
<i>Planning</i> —Planning the steps needed to implement the solution.	“List out the tasks that need to be done based on the requirements of the problem/client and the technology stack available.” (P51)
<i>Implementing</i> —Implementing a particular solution.	“Build an implementation from the proof of concept.” (P52)
<i>Updating Implementation</i> —Trying a new implementation or updating an existing one based on previous implementation attempts.	“If there is a problem with the solution that’s implemented, go to step 1 with what you learned by implementing the solution.” (P29)
<i>Deploying</i> —Releasing the solution to the public.	“Implement your solution...and deploy it into a development environment.” (P28)
Verifying (×1)	
<i>Reviewing</i> —Having others review and provide feedback to the solution.	“Have others review your implementation proposal (over coffee can help).” (P28)
<i>Testing</i> —Testing the implementation for functionality and potential defects.	“Test your implementation until you find most of the bugs you can and your teams agree to release to prod.” (P52)
Updating Knowledge (×4)	
<i>Researching</i> —Learning more about the problem or potential solutions.	“Search online whether other libraries support your use case.” (P6)

Expertise relating to deeply understanding the vision of the project (e.g., knowledgeable about customers and business) was most frequently referenced. Expertise on evaluating the pros and cons of a solution (e.g., makes tradeoffs) was also mentioned. Participants also frequently described forms of technical expertise (e.g., knowledgeable about tools and building materials).

Expertise that was less commonly cited in strategies largely related to aspects of the software and designs (e.g., attentive to details). Expertise about teammates or the company (e.g., knowledgeable about the people and organization) and responding to changing problem contexts (e.g., updates their mental models) were also less referenced.

V. THREATS TO VALIDITY

1) *Internal validity*: Strategy content may have been influenced by the study authors since they were initially recorded

by them. To address this, the authors asked participants to review and confirm the strategy multiple times to ensure the strategy was accurate and written as the participants wished. Participants could also directly make changes to the strategy upon request.

The authors could have confirmation biases that developer actions must follow normative theories of the software development life cycle, which could influence the generation of action codes. We reduced this threat by achieving consensus on each code applied in our qualitative analysis. Future studies using other methods, such as contextual inquiries of implementation design decision-making, could address this bias.

Additionally, memory bias could have introduced inaccuracies in participants’ recall on past decisions, considerations, and strategies. We reduced this bias by asking participants to record decisions, considerations, and actions that occurred in the past five days in both the survey and interview.

TABLE V
MEDIAN POSITION OF ACTIONS IN PARTICIPANTS' STRATEGIES AND PERCENT OF STRATEGIES CONTAINING ACTIONS.

Action	Median Position	% Strategies w/ Action
Providing Context	1.5	12.5%
Researching	2	75.0%
Defining Requirements	2	81.3%
Brainstorming	3	62.5%
Estimating	3	25.0%
Evaluating	4	81.3%
Choosing	5	81.3%
Planning	6	25.0%
Proof-of-Concept	6.5	31.3%
Updating Requirements	7	43.8%
Implementing	7	68.8%
Reviewing	8	43.8%
Testing	9	31.3%
Updating Implementation	9	18.8%
Deploying	10.5	12.5%

TABLE VI
FREQUENCY OF DEVELOPER EXPERTISE FROM LI ET AL. [1] IN PARTICIPANTS' STRATEGIES.

Expertise (quoted from Li et al. [1])	Count
Decision Making	
Knowledgeable about customers and business	56
Sees the forest and the trees	45
Knowledgeable about tools and building materials	39
Knowledgeable about their technical domain	38
Knowledgeable about engineering processes	31
Models states and outcomes	24
Handles complexity	24
Knowledgeable about people and the organization	13
Updates their mental models	8
Software & Designs	
Carefully constructed	26
Fitted	11
Evolving	10
Attentive to details	9
Anticipates needs	5
Creative	5
Long-termed	2
Elegant	2

Participants could have been primed in the implementation design decisions they reported from the survey and interview examples. We reduced this threat by providing a short, diverse set of examples to show the breadth of the phenomenon.

Study participants may have misunderstood the wording of the questions. To reduce this threat, we piloted the survey and interview with software developers and study team members and asked for feedback on clarity. We also performed pilots on the first 15 survey responses for data quality.

2) *External validity*: Any small-scale empirical study has generalizability issues [61]. To address this, we sampled Reddit users across a diverse set of subreddits. Our sample represents diverse geographic regions, engineering organization sizes, roles, and amounts of relevant experience. Additionally, we also collected the data on decisions and considerations using two methods, which we used to corroborate results.

Empirical studies can also suffer from selection bias. The subreddits we recruited from may be homogeneous due having to common interests, so some programming expertise was not represented in our study (e.g., game development). Our survey also limited representation to regions where English is a primary language due to the survey being written in English. We addressed this threat by ensuring the survey was as short as possible, accurately advertising the survey's length, and providing incentives for participating in the interview.

One common issue for programming strategies is that they may not generalize due to defects or having a narrow scope [48]. To address this concern, we asked participants to test and fix their strategy and verify whether their strategy was generalized. The study authors were familiar with writing high-quality and generalized strategies.

3) *Construct validity*: Since the participants' decisions, considerations, and strategies were self-reported, there could be inconsistencies in what participants report doing versus what they actually did. They may have forgotten to explicitly mention actions or misremembered the process they used.

VI. DISCUSSION & FUTURE WORK

Our findings overlap some with prior work (e.g., [1], [14], [33], [62], [63]). In this section, we discuss them in relation to implementation design decisions. This produces several implications, which we elaborate on.

Our results suggest that implementation design decisions are shaped by higher levels of design (e.g., requirements and architecture). Also, a developer's decisions can directly and intentionally shape these higher level concerns. Thus, interpreting requirements throughout implementation is key to making these decisions. Requirements appeared in the decisions (e.g., behaviors), considerations (e.g., future requirements), and actions (e.g., defining requirements) of software developers. It also was the most frequently cited form of expertise (e.g., knowledgeable about customers and business) in strategies. This highlights the perspective that requirements engineering is an ongoing process throughout implementation and maintenance. Depending on how much control the developer had, they could re-interpret or completely change

requirements, suggesting that understanding *how* to update requirements to match dynamic contexts could be a software engineering skill. This contrasts the notion of requirements being set prior to implementation. Rather, it supports prior work stating that requirements can be iterated upon through prototyping [55]–[57].

Next, we find that maintainability is a major theme in implementation design decisions. It appeared in the decisions (e.g., reuse) and considerations (e.g., extensibility, reusing resources) as it reduced workload, cognitive load, and technical debt. This suggests that software developers may need to develop a sense of how to anticipate maintenance effort and develop knowledge on how to manage and reduce debt, such as separation of concerns and modularity.

We also find that the process to make implementation design decisions is both an art and a science. Across different developers and problems, there were strong commonalities in the considerations and strategy structure. However, each strategy was unique—prior work has shown that software development teams also follow their own individual processes for early stage software design work [43]. This implies that making implementation design decisions has a common structure. Yet, it requires expert judgment developed from experience with similar problems to know when to deviate from it for the given use case. Furthermore, it suggests this form of design expertise is both systematic and opportunistic, which has been observed in prior work [28], [64]. One source of opportunism is when similar types of actions are repeated in different parts of the strategy. This suggests that implementation design decisions require rounds of iteration in different stages, especially in implementing and defining the problem space.

Finally, our results suggest that some forms of expertise are more implicit, while others are more tacit. Explicit programming strategies are a form of explicit programming knowledge [47]. Participants’ strategies largely referenced decision-making expertise, implying that it could be explicit knowledge. Meanwhile, expertise on developing software and designs was referenced noticeably less frequently in participants’ strategies but instead overlapped with our enumerated considerations. This suggests that this form of knowledge is more tacit and is applied in the moment of programming problem-solving.

These implications affect software engineering researchers, educators, and practitioners. We describe how our findings apply to them and provide opportunities for future work.

A. Educators

Our findings have implications on how to teach programming and software design. While teaching software engineering, educators could consider providing open-ended projects, as suggested by Offutt and Baral [65]. This would provide students with opportunities to make various types of implementation design decisions. Further, these projects could span for a longer duration to teach students how their decisions shape software maintenance. Educators could scaffold students’ problem-solving process by authoring their own

explicit programming strategies on how to make implementation design decisions. Educators could also use the list of considerations from Table III as a checklist for students to follow when evaluating candidate solutions.

B. Software Engineers

Our findings can help novice engineers to make better implementation design decisions. Since iteration is an important part of making implementation design decisions, less experienced engineers could work prototyping into their regular practice and become accustomed to learning from small-scale failures through prototyping. Managers or mentors could reinforce this learning environment by encouraging this practice.

Additionally, novices could also use the list of considerations as a checklist to help make decisions. More senior engineers could extend our list of considerations by writing their own definitions and heuristics. They could also add their own considerations to teach less experienced team members.

C. Researchers


This work raises questions about our understanding of software design. Previous work viewed software design as a sociotechnical process (e.g., [41]), a set of habits (e.g., [33]), or as high-level code structure (e.g., [15]). Our work extends this knowledge by focusing on the cognitive process involved in software design. We view software design as a decision-making exercise, following prior work (e.g., [8], [14]).

There are several directions that require additional study. Future work could study each of the the consideration codes to discover how developers estimate them and how they compare considerations against one another. This could help develop automated metrics or tools to aid developers’ decision-making. Additionally, future work could examine how experts’ decision-making processes differ than that of novices’, especially in the strategy structure and considerations. This could help better understand the attributes that relate to effective decision-making processes and advance understanding of software engineering expertise.

DATA AVAILABILITY

Our supplemental materials are available on Figshare [45]. Data includes the codebooks for **RQ1**, **RQ2**, and **RQ3**; the plain text, action code, and action category representations of participants’ strategies; the survey instrument; and the interview protocol.

ACKNOWLEDGMENTS

We thank our survey and interview participants for their insight and Soham Pardeshi, Lilian Liang, Nimit Johri, and Tobias Dürschmid for their feedback. We give special thanks to Mei , an outstanding canine software engineering researcher, for providing support and motivation throughout this study.

This work was supported by the National Science Foundation under grants 1539179, 1703734, 1703304, 1836813, 1845508, 2031265, 2100296, 2122950, 2137834, 2137312, and by unrestricted gifts from Microsoft, Adobe, and Google.

REFERENCES

- [1] P. L. Li, A. J. Ko, and J. Zhu, "What makes a great software engineer?" in *IEEE/ACM International Conference on Software Engineering*, 2015, pp. 700–710.
- [2] A. Tang, M. H. Tran, J. Han, and H. van Vliet, "Design reasoning improves software design quality," in *International Conference on the Quality of Software Architectures*, 2008, pp. 28–42.
- [3] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *IEEE/ACM International Workshop on Program Comprehension*, 2002, pp. 271–278.
- [4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.
- [5] R. Verdecchia, P. Kruchten, and P. Lago, "Architectural technical debt: A grounded theory," in *European Conference on Software Architecture*, 2020, pp. 202–219.
- [6] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: How and why software developers use drawings," in *SIGCHI Conference on Human Factors in Computing Systems*, 2007, pp. 557–566.
- [7] M. Petre, "UML in practice," in *IEEE/ACM International Conference on Software Engineering*, 2013, pp. 722–731.
- [8] H. van Vliet and A. Tang, "Decision making in software architecture," *Journal of Systems and Software*, vol. 117, pp. 638–644, 2016.
- [9] A. Tang, M. Razavian, B. Paech, and T. Hesse, "Human aspects in software architecture decision making," in *IEEE International Conference on Software Architecture*, 2017, pp. 107–116.
- [10] A. Shahbazian, S. Karthik, Y. Brun, and N. Medvidovic, "eQual: informing early design decisions," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1039–1051.
- [11] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer, "Tool support for architectural decisions," in *Working IEEE/IFIP Conference on Software Architecture*, 2007, pp. 4–4.
- [12] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson, 1995.
- [13] K. Beck, R. Crocker, G. Meszaros, J. O. Coplien, L. Dominick, F. Paulisch, and J. Vlissides, "Industrial experience with design patterns," in *IEEE International Conference on Software Engineering*, 1996, pp. 103–114.
- [14] P. Ralph and E. Tempero, "Characteristics of decision-making during coding," in *International Conference on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–10.
- [15] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.
- [16] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.
- [17] T. Besker, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work," *Journal of Systems and Software*, vol. 156, pp. 41–61, 2019.
- [18] N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman, "The most common causes and effects of technical debt: First results from a global family of industrial surveys," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.
- [19] —, "Supporting analysis of technical debt causes and effects with cross-company probabilistic cause-effect diagrams," in *IEEE/ACM International Conference on Technical Debt*, 2019, pp. 3–12.
- [20] G. Ruhe, "Software engineering decision support—a new paradigm for learning software organizations," in *International Workshop on Learning Software Organizations*, 2002, pp. 104–113.
- [21] C. Becker, D. Walker, and C. McCord, "Intertemporal choice: Decision-making and time in software engineering," in *IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering*, 2017, pp. 23–29.
- [22] S. Baltes and S. Diehl, "Towards a theory of software development expertise," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 187–200.
- [23] P. L. Li, A. J. Ko, and A. Begel, "What distinguishes great software engineers?" *Empirical Software Engineering*, vol. 25, no. 1, pp. 322–352, 2020.
- [24] A. Aurum and C. Wohlin, "The fundamental nature of requirements engineering activities as a decision-making process," *Information and Software Technology*, vol. 45, no. 14, pp. 945–954, 2003.
- [25] P. Berander and A. Andrews, "Requirements prioritization," in *Engineering and Managing Software Requirements*. Springer, 2005, pp. 69–94.
- [26] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, "Decision-making techniques for software architecture design: A comparative survey," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–28, 2011.
- [27] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 2003.
- [28] C. Zannier, M. Chiasson, and F. Maurer, "A model of design decision making based on empirical results of interviews with software designers," *Information and Software Technology*, vol. 49, no. 6, pp. 637–653, 2007.
- [29] J. Stylos and B. Myers, "Mapping the space of API design decisions," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007, pp. 50–60.
- [30] R. Dharani, *Web API Design: Crafting Interfaces that Developers Love*. Independently published, 2017.
- [31] "Swift.org—API design guidelines," 2022, retrieved December 20, 2022 from <https://www.swift.org/documentation/api-design-guidelines/>.
- [32] "API design," 2022, retrieved December 20, 2022 from <https://martinfowler.com/tags/API%20design.html>.
- [33] M. Petre and A. van der Hoek, *Software Design Decoded: 66 Ways Experts Think*. MIT Press, 2016.
- [34] J. Bloch, *Effective Java*. Addison-Wesley Professional, 2008.
- [35] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.
- [36] —, *Just Enough Software Architecture: A Risk-Driven Approach*. Prentice Hall, 2018.
- [37] G. Fairbanks, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Marshall & Brainerd, 2018.
- [38] A. J. Ko and P. K. Chilana, "Design, discussion, and dissent in open bug reports," in *iConference*, 2011, pp. 106–113.
- [39] J. Brunet, G. C. Murphy, R. Terra, J. Figueiredo, and D. Serey, "Do developers discuss design?" in *Working Conference on Mining Software Repositories*, 2014, pp. 340–343.
- [40] A. Mahadi, N. A. Ernst, and K. Tongay, "Conclusion stability for natural language based mining of design discussions," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–42, 2022.
- [41] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek, "Supporting informal design with interactive whiteboards," in *SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 331–340.
- [42] S. Baltes and S. Diehl, "Sketches and diagrams in practice," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 530–541.
- [43] B. Sharif, N. Dragan, A. Sutton, M. L. Collard, and J. I. Maletic, "Identifying and analyzing software design activities," in *Software Designers in Action: A Human-Centric Look at Design Work*. Chapman and Hall/CRC, 2013, pp. 153–174.
- [44] M. K. Scheuerman, K. Spiel, O. L. Haimson, F. Hamidi, and S. M. Branham, "HCI guidelines for gender equity and inclusivity," in *UMBC Faculty Collection*, 2020.
- [45] J. T. Liang, M. Arab, M. Ko, A. J. Ko, and T. D. LaToza, "Supplemental materials to "A qualitative study on the implementation design decisions of developers";" 2023, available at <https://doi.org/10.6084/m9.figshare.21820140>.
- [46] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.
- [47] T. D. LaToza, M. Arab, D. Loksa, and A. J. Ko, "Explicit programming strategies," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2416–2449, 2020.
- [48] M. Arab, T. D. LaToza, J. Liang, and A. J. Ko, "An exploratory study of sharing strategic programming knowledge," in *SIGCHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–15.
- [49] M. Arab, J. Liang, Y. Yoo, A. J. Ko, and T. D. LaToza, "HowToo: A platform for sharing, finding, and using programming strategies," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2021, pp. 1–9.

- [50] D. Ford, T. Zimmermann, C. Bird, and N. Nagappan, "Characterizing software engineering work with personas based on knowledge worker actions," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2017, pp. 394–403.
- [51] E. K. Smith, C. Bird, and T. Zimmermann, "Beliefs, practices, and personalities of software engineers: A survey in a large software company," in *International Workshop on Cooperative and Human Aspects of Software Engineering*, 2016, pp. 15–18.
- [52] D. Hammer and L. K. Berland, "Confusing claims for data: A critique of common practices for presenting qualitative research on learning," *Journal of the Learning Sciences*, vol. 23, no. 1, pp. 37–46, 2014.
- [53] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.
- [54] M. B. Miles and A. M. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE Publications, 1994.
- [55] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.
- [56] F. Paetsch, A. Eberlein, and F. Maurer, "Requirements engineering and agile software development," in *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2003, pp. 308–313.
- [57] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Conference on the Future of Software Engineering*, 2000, pp. 35–46.
- [58] L. MacLeod, M.-A. Storey, and A. Bergen, "Code, camera, action: How software developers document and share program knowledge using YouTube," in *IEEE International Conference on Program Comprehension*, 2015, pp. 104–114.
- [59] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow," *Georgia Institute of Technology Technical Report*, vol. 11, 2012.
- [60] C. D. Hardin and M. Berland, "Learning to program using online forums: A comparison of links posted on Reddit and Stack Overflow," in *ACM Technical Symposium on Computing Science Education*, 2016, pp. 723–723.
- [61] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative Inquiry*, vol. 12, no. 2, pp. 219–245, 2006.
- [62] S. Balaji and M. S. Murugaiyan, "Waterfall vs. V-model vs. Agile: A comparative study on sdlc," *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.
- [63] M. Petre, A. van der Hoek, and D. S. Bowers, "Software design as multiple contrasting dialogues," in *Psychology of Programming Interest Group 30th Annual Conference*, 2019.
- [64] S. P. Davies, "Characterizing the program design activity: Neither strictly top-down nor globally opportunistic," *Behaviour & Information Technology*, vol. 10, no. 3, pp. 173–190, 1991.
- [65] J. Offutt and K. Baral, "Designing divergent thinking, creative problem solving exams," in *ACM/IEEE International Conference on Software Engineering: Software Engineering Education and Training*, 2022, pp. 82–89.