Practical Knowledge Barriers in Professional Programming

Kyle Thayer

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Amy J. Ko, Chair

Katharina Reinecke

Mike Ernst

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Practical Knowledge Barriers in Professional Programming

Kyle Thayer

Chair of the Supervisory Committee:
Title of Chair Amy J. Ko
Department of Chair

More and more jobs are being created that involve programming. Parts of the practical knowledge needed to get these jobs and succeed in these jobs is not taught in traditional university computer science departments. A lack of this knowledge can be a barrier to successfully programming professionally. In order to learn more about these knowledge barriers, I (along with my co-authors) have taken on three projects to understand these barriers and how people can get around them: 1) we studied the barriers faced by coding bootcamp students, finding non-knowledge based barriers such as credentials, personal costs, as well as knowledge barriers such as how to pass a technical interview, and using available resources to learn how to use application programming interfaces (APIs); 2) Inspired by the barriers to learning APIs in the first study, we developed a theory of what knowledge is actually needed to work with APIs, including API usage patterns; 3) Inspired by the need to understand API usage patterns in the second study, we developed a data structure to represent API usage patterns and an algorithm to extract part of the structure. For people who want to address these barriers at a systemic level, this work can guide them in designing bootcamps or other programs to address these needs, or in designing to help people learn and use APIs. For people who want to become programmers, this work can guide them in deciding whether a bootcamp will meet their needs and knowing what other needs they may have, and future systems based on our API work may help them learn and use APIs.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Number Page

ACKNOWLEDGMENTS

I would like to thank my advisors Amy Ko and Katharina Reinecke or their support and help. I'd also like to thank the other co-authors on my papers Philip Guo and Sarah Chasins for their help. I'd like to thank Benji Xie, Jacob O. Wobbrock, Nigini A. Oliveira, Daniel Epstein, Trevor Perrier, Melissa Hovik, Lauren Bricker, Fahad Pervaiz, Amanda Swearngin, Eunice Jun, William Tressel, Kurtis Heimerl, Alannah Oleson, Harrison Kwik, Leanne Hwa, Alex Tan, and Rahul Banerjee for their feedback and help. Finally I'd like to thank Kristen Thayer for her support, help, and encouragement. Portions of this material are based upon work supported by Microsoft, Google, Adobe, and the National Science Foundation under Grant No. 1539179, 1314399, 1240786, and 1153625.

DEDICATION

to Kristen, Kendra, and Kaylee

Chapter 1

INTRODUCTION

1.1 Barriers in Professional Software Engineering

Demand for software developers is growing (e.g., an expected 17% in the US between 2014 and 2024 [14]). These jobs can provide significant income and opportunities beyond what many other careers provide [101, 138].

People face numerous barriers when trying to get hired for programming jobs and trying to succeed in them, such as expectations of college degrees as well as racial and gender biases [7, 97, 15] (see also chapter 2). Some of these barriers are practical knowledge barriers, that is, they are based on knowledge needed to perform actions like finding job opportunities, writing computer code, or communicating with teammates. When trying to get hired to do programming professionally, people may face requirements in knowing programming languages and data structures, as well as knowing how to work with others [44, 15]. When joining a new programming team, "employees attempt to master immense amounts of material in a short amount of time" [8]. When modifying a project, programmers need to understand the project at different levels of abstraction [120], and need to understand the specific domain and tools of their project [67]. The knowledge barriers at each of these stages may be based on technical knowledge (general or project-specific), as well as other kinds of knowledge, such as know about their team environment and business environment, like who is working on what task [57, 35], who has what expertise [67, 54, 57], what business needs does the project fit with [67, 54], and how to fit in with the culture of the team and organization [44] (though culture fit can be an avenue for bias against marginalized groups[45, 46]). Parts of this knowledge are not covered in traditional undergraduate computer science programs and must be learned on the job [15, 3, 9]. Still, these barriers are not all fully understood, and people still run into these barriers, sometimes failing to get jobs [101] or struggle once in them [9].

1.2 Investigating Practical Knowledge Barriers and How to Overcome Them

In order to explore further these practical knowledge barriers and how they can be overcome I, along with my co-authors, ask the following two research questions:

- RQ1: What practical knowledge barriers are faced by professional programmers and those seeking to become professional programmers?
- RQ2: How can these barriers be overcome?

We provide new insights on these questions through a series of three projects:

- A qualitative study of coding bootcamp students, investigating the barriers students face in trying to get programming jobs through bootcamps, as well as barriers they face in the bootcamps (Chapter 2) [132]. This study focused on many types of barriers and not just knowledge barriers, so for example we found many barriers in both bootcamps and job hiring processes that also have shown up other programming education contexts, such as gender and racial biases (though some bootcamps did more to overcome these). We also learned more about knowledge barriers in the software industry, and how bootcamps helped (or sometimes failed to help) students gain this knowledge in order to get hired and succeed in software jobs. This knowledge included how to network, how to work on programming teams, and how to make sense of available online documentation for Application Programmer Interfaces (APIs). This last set of knowledge (learning APIs from available resources) was in particular discussed by many bootcamp students, including how bootcamps (with varying levels of success) tried to address this barrier. This inspired us to ask what was this knowledge about APIs that bootcamp students were trying to discover so that they could successfully program with the APIs. We answered this through our next project:

- We developed a theory of the knowledge needed to work with APIs (Chapter 3) [131]. We defined three core types of knowledge: 1) domain concepts and the terminology used in the API, 2) API usage patterns, and 3) execution facts about the API's runtime behavior. We then completed two studies to provide supporting evidence to our theory. After considering these three types of knowledge in relation to what resources were available to developers and what how we could feasibly address the needs of developers, we decided to focus on API usage patterns (official API documentation is often contains some API runtime behavior information and domain knowledge we considered to broad to easily address). In order to investigate how to reduce barriers to finding and learning API usage patterns our next project looked at building representations of API usage patterns:

- We defined a data structure and created an algorithm to represent and extract API usage patterns in the form of templates (Chapter 4). We extracted these templates from online code repositories. We then present a series of potential use cases for for the data structure, and the results of a preliminary study showing how templates can aid developers.

Through these projects we have gained additional insight into the knowledge barriers people face in becoming professional programmers and succeeding in those jobs, we have seen how people use bootcamps to gain some of that knowledge, and we have begun work on automatically discovering some of that knowledge in order to aid future programmers.

1.3 Contribution

In this dissertation we present a template for identifying, understanding, and addressing practical knowledge barriers in professional programming, allowing us to begin addressing barriers faced by a more diverse group of people and at scale. We do this by finding barriers faced by a diverse set of people entering the software industry through the new, alternate route of coding bootcamps, we understand one barrier through building an explanatory theory of API knowledge, and we set up how to address part of that barrier at scale through API usage templates.

In chapter 2, we contribute:

- The first peer-reviewed publication to look at coding bootcamps, which have a different focus and often teach a different set of practical programming knowledge than CS undergraduate programs.

- By studying a new, alternate pathway into the software industry, we get us a new perspectives on barriers in professional programming, and that helped us both identify the barriers faced by this group of people, and compare them against the barriers faced in undergraduate computer science degrees.

In chapter 3, we contribute:

- An explanatory theory of API knowledge (one of the barriers highlighted in chapter 2), explaining the role of three parts of knowledge in working with API code and the relations between those parts.

- Seven hypotheses based on our theory, and two studies to test them. We found mixed evidence for our theory, pointing to the need of further testing and refinement of the theory.

In chapter 4, we contribute:

- A new, flexible data structure (API usage templates) to represent one of the components of knowledge from chapter 2.

- A set of 12 scenarios of API use where that data structure could be beneficial and provide novel solutions.

- An algorithm to automatically extract part of the API usage template data structure from example code using the API, which would allow for scalable creation of API usage templates.

- A preliminary study finding that developers found an API task easier with access to API usage templates that were based on results from our algorithm.

In total, this work demonstrates a series of steps that help us address the knowledge barriers faced by diverse people using understudied training paths (in this case, coding bootcamps) in trying to enter the software industry, and we begin work needed to be able scale solutions for the tens of thousands of people facing these barriers.

Chapter 2

PROJECT 1: BARRIERS FACED BY CODING BOOTCAMP STUDENTS

2.1 Introduction

In response to the increasing market demand for professional software developers [14], more people are graduating from undergraduate computer science (CS) programs (Figure 2.1), while others are learning software development from online tutorials, Massive Open Online Courses (MOOCs), and now fast-paced coding bootcamps [98, 1]. Coding bootcamps have grown rapidly in the US and Canada since 2013 (Figure 2.1) and serve a different population than undergraduate programs [101].

In spite of the growth of bootcamps, we know little about how bootcamps relate to the barriers of programming professionally, and other barriers bootcamp students may face. Prior reports on bootcamps have only focused on the logistics of bootcamps [101, 117], or the demographics and success rate of their graduates [102], and little peer-reviewed research has been published since this work was published [15, 135]. Research in computing education and career change suggests several barriers bootcamp students might face in learning programming. For example, in various computing education contexts (high schools, colleges and universities), societal pressures cause divisions in who is encouraged to use computers and who is made to feel welcome in computing classes, in particular excluding female, black and latino/a students [71, 70, 11, 17]. In computing classes, students face stereotypes of what it takes to be a "real" programmer [71, 70, 17, 65] and those who feel belonging, comfort and confidence are better able to succeed [144, 116]. Similarly, people changing careers face barriers in getting jobs that may be based on confidence, gender, age, and educational level [16, 103, 85] while also facing pressures due to lost income and the effect of lost income on

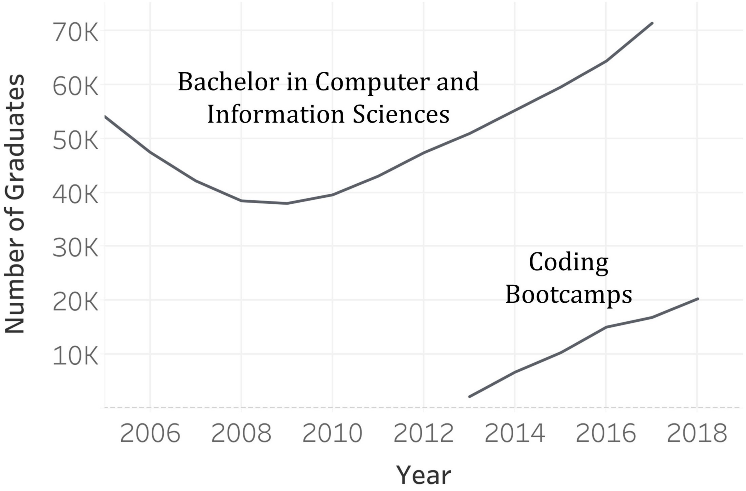


Figure 2.1: Yearly US bachelor in computer and information science graduation rate (by end of academic year) [83], along with yearly coding bootcamp actual and projected graduation rate in the US and Canada [102].

family [103, 85].

We hypothesized that the barriers faced in other computing education and career change contexts would also be faced by bootcamp students as they went through bootcamps and sought jobs in the software industry. We therefore interviewed 26 current and former bootcamp students to ask about their stories and the barriers they faced. Our participants represented eight bootcamps and a range of trajectories and stages, from early in a bootcamp to having finished. After considering frameworks for learning [141, 63, 48] and career change [103, 121], we decided to analyze our data with *Communities of Practice* [141] and concepts from the career change literature.

This work has previously been published [132].

2.2 Related Work

2.2.1 Communities of Practice

Communities of practice are "groups of people who share a concern or a passion for something they do and learn how to do it better as they interact regularly" [142]. These communities can range from formal such as a work team in an office, or informal such as a clique of students. We considered each bootcamp as its own community of practice and software industry jobs as communities of practice tied together in a *constellation of practice* [141].

The *Communities of Practice* framework [141] provided several useful concepts for analyzing and framing our results.[1] Communities of practice have community defined *boundaries* (both *formal* like member lists and *informal* like specialized jargon) which define what is *inside*, *outside* and at the *periphery* of the community. For joining a community, these boundaries must be *negotiated* with the community and are one type of barrier to entry. An individual's relationship with a community evolves over time as part of their *learning trajectory* [141]. As individuals belong to multiple communities of practice simultaneously, they face conflicting meanings and practices. The *Communities of Practice* framework has been used to understand the design of schools and businesses [141], apprenticeships [63], career change [143], involvement in Open Source communities [147], identity formation [93], and course design [38].

2.2.2 Barriers in Career Change

Another concept we used for analyzing barriers was *personal obstacles*,[2] which came from the career change literature. People changing careers face *personal obstacles* that include age, gender, financial considerations around temporary lost income (especially when they had dependent spouses or children), education level, personality and confidence [103, 16, 85].

[1] Since Wenger[141] rarely includes formal definitions of his terms, we provide our own.

[2] *Personal obstacles* is our term combining "personal factors" and "obstacles" [103, 85].

2.2.3 Barriers in Computing Education

Previous studies on barriers in computing education have mostly focused on barriers students face in choosing and continuing CS studies in high school, college and university settings. In *Unlocking the Clubhouse* [71], Margolis and Fisher found barriers for women in undergraduate CS that included admissions (*formal boundary*), gender divides in computer use from a young age, stereotypes of who a "real" programmer is (e.g., anti-social), expected background experience and a belief in a "natural" ability to understand computers (*informal boundaries*). They also found women faced barriers of lost confidence and lack of social support (*personal obstacles*). *Stuck in the Shallow End: Education, Race and Computing* by Margolis, Estrella, et al. [70] examined the racial gap in high school CS, finding barriers that included lack of access to classes (*formal boundary*), cultural expectations on who the classes were for, feelings of isolation in classes, divisions within classes between those who "have it or don't have it" (*informal boundaries*), and lack of social support (*personal obstacle*). Additional studies found participation and success in computing programs depended on background experience [11, 144], comfort level [144], sense of belonging and stereotypes (disproportionately negatively affecting women) [11, 17, 65, 93], view of self as an "insider" [116], and believed role of luck [144].

In addition to these studies, there have been posters, marketing reports and commissioned reports on bootcamps [102, 101, 117, 69]. In the US and Canada in 2018, bootcamps had an average tuition of $11,900 and length of 14.3 weeks [102]. Bootcamp graduates were diverse in backgrounds (57% had previous full-time employment and 35% had never programmed before) and diverse in gender (34% were female, compared to the 19% of CS graduates) [102]. A report on international bootcamps briefly mentioned students may face *formal boundaries* (admissions, payment, and graduation), *informal boundaries* (gender), and *personal obstacles* (intensity, time, location, and family support), but it didn't provide details [117]. In the time this work was first published, Burke et. al. interviewed employers about coding bootcamps and undergraduate computer science degrees, as well as instructors in those two types of programs [15]. They found that employers looked for both "hard skills" that consisted of technical knowledge and "soft skills" like adaptability, team work, and creativity. The employers and instructors viewed differences in what bootcamps and CS programs offered, with CS programs offering more theoretical knowledge and algorithms, and bootcamps providing more knowledge of the latest technologies and more practice with team work and practical problem solving.

2.3 Method

To study barriers in bootcamps and the software industry, we interviewed current and former students of bootcamps, focusing on bootcamps in the Puget Sound area (Washington, USA). We defined coding bootcamps as non-university programs that offered full-time, in-person, short-term (months-long) software development training. This excluded weekend, night, and part-time classes, strictly online bootcamps and any program that takes more than one year. We also excluded bootcamps that were not primarily targeted for software engineering jobs (e.g., data science, UX).

We found an initial group of bootcamp students through personal connections, LinkedIn, and a weekend programming class. From there we used stratified snowball sampling to find a range of bootcamp students. We focused on recruiting participants from different bootcamps, at different stages (*in bootcamp*, *post-bootcamp*, *job hunting*, *in job*, *no longer searching for a job*), as well as diversity in race and gender. We conducted 26 interviews and had at least two students from each of six full-time bootcamps in the Puget Sound area: Ada Developers Academy, Code Fellows, Coding Dojo, Dev Bootcamp, Galvanize, and General Assembly, as well as one student each from two out of state bootcamps. We had at least ten females and eight males.[3] We had students who were Black, White, Asian, Latino/a, and at least five who were more than one race or ethnicity. The youngest participant (who we know the age of) started a bootcamp at age 18, and the oldest at 39. We also interviewed students who

[3] We did not ask for demographic information in some interviews. For those, we counted any statements participants made which stated or implied their demographics.

identified as straight and as gay.

We developed a semi-structured interview protocol (Appendix A) consisting of twenty-five questions divided into four sections: *background*, *deciding to attend a bootcamp*, *changes in views and goals*, and *how they perceive their experience in relation to others'*. We piloted and refined the questions with the help of someone changing careers into the software industry, though not through a bootcamp. The length of the interviews ranged from 24 to 94 minutes with a median length of 43 minutes. After completing the interviews, we transcribed them, removed personally identifiable information, and deleted the recordings.

From the interviews, we created chronological *coding bootcamp trajectories* and *software development trajectories* for each participant. We categorized the pieces of each trajectory by how they related to the community of practice. We then coded all discussions of *formal boundaries* (e.g., admissions, graduation, and hiring), *informal boundaries* (e.g., fitting in, unstated expectations, and group dynamics) with respect to the two communities of practice. We also did this for discussions of *personal obstacles*, which we defined as obstacles to negotiating community boundaries that were not concerns of the community (e.g., personal financial burdens and relational costs). We then synthesized the results in each category.

2.4 Results

Because the coding bootcamp students we talked to viewed entering the software industry as their high-level goal (with bootcamps as a means to that end), we first discuss students' software industry trajectories and then discuss how students' bootcamp experiences related to these trajectories. We include quotes throughout, selectively omitting identifying information to preserve anonymity, and making minor edits for clarity. Any emotions reported are those explicitly stated by participants.

2.4.1 Participant Learning Trajectories

Software Industry Learning Trajectories

For each participant, we mapped each step of their *software industry trajectories* chronologically using the following four levels of involvement in the software industry: *unrelated activities* (e.g., other education, jobs), *preparation to enter the software industry* (e.g., classes, bootcamps, building a portfolio), *partial employment as a software developer* (e.g., contracts, internships), and *full employment as a software developer* (the stated goal of all participants).[4] Figure 2.2 shows the variety of our participants' trajectories. For example, participants P1, P2 and P3 went from unrelated education and careers into full employment while P26 returned to their former career after failing to get employment as a software developer. Nineteen participants took online courses, ten took separate in-person classes, and P5, P8 and P13 had degrees in CS before starting a bootcamp. Participants P18, P22, and P23 attended more than one bootcamp and participants P13, P14, P15, and P24 did not finish a bootcamp and had no plans to. Participants P19, P20 and P21 went to a bootcamp that had a built-in internship.

Coding Bootcamp Learning Trajectories

[5] Since the stated goal of coding bootcamps was to prepare students for entering the software industry, students' trajectories through bootcamps were a part of their trajectories into the software industry. Because of this, most students' bootcamp trajectories appear much like the first half of their software industry trajectories, so we do not show them here. Some students took actions to prepare for the software industry before starting a bootcamp, which incidentally also helped prepare them for their bootcamps. Other students prepared specifi-

[4] Mapping the learning trajectories of our participants into these categories was mostly straightforward, though there were occasions where chronology was unclear.

[5] In discussing bootcamps, it should be noted that bootcamps are new and changing rapidly, with students mentioning significant changes in courses, content, and social dynamics between cohorts or even within their own journey through a bootcamp.

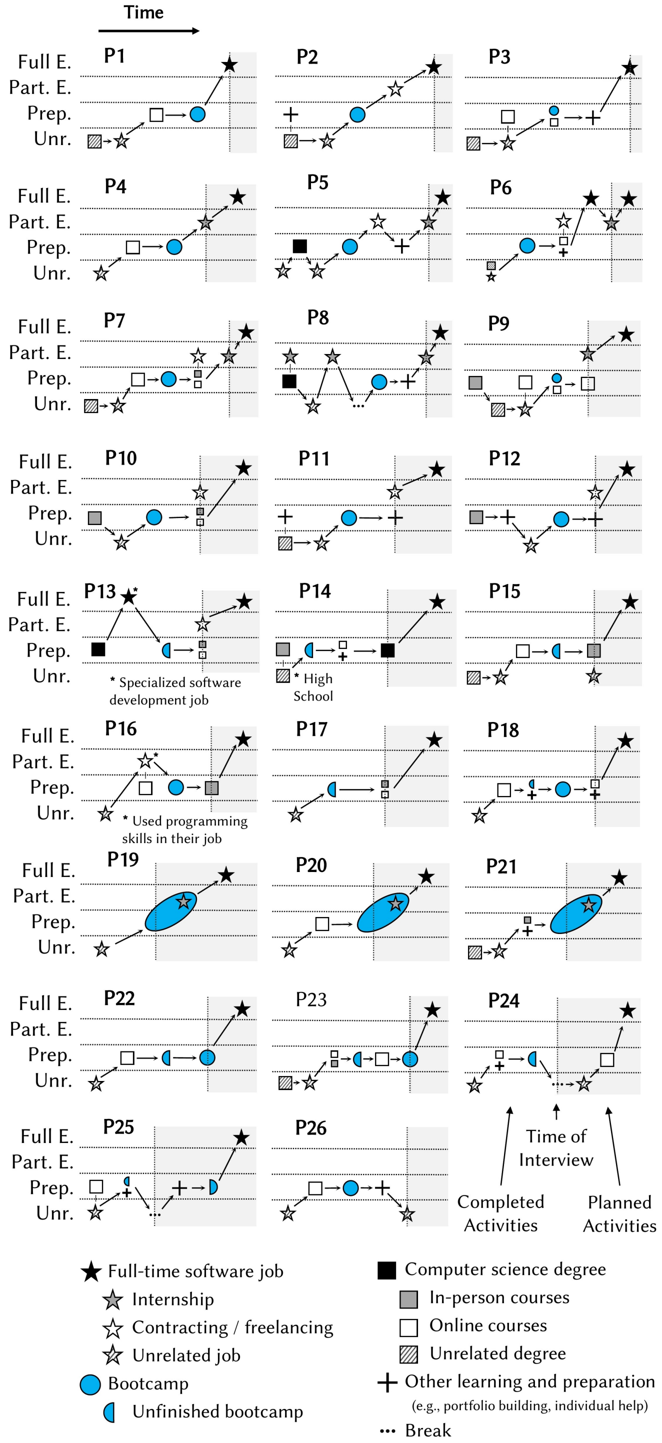


Figure 2.2: Software industry trajectories for all participants. Each graph show a participant's chronological activities in four levels of increasing software industry involvement: *unrelated* (Unr.), *preparation* (Prep.), *partial employment* (Part. E.), and *full employment* (Full E.). Participants are sorted by industry involvement at time of interview.

cally for bootcamps, including P22, who attended the intro section of one bootcamp in order to improve their chances of getting into another bootcamp. While attending bootcamps, some students felt the tests, assignments and even bootcamp graduation did not align with their software industry trajectory. Because of this perceived misalignment, P25 took a break from the bootcamp to study more, P13 quit their bootcamp, and P6 suggested ignoring some bootcamp content and deadlines. After graduating from a bootcamp, some students continued to be involved through residencies (free space and time for building portfolios), paid TA positions, and alumni networks.

2.4.2 Barriers in the Software Industry

Having seen students' diverse software industry and bootcamp trajectories, we now focus on the boundaries and personal obstacles they faced in the context of their software industry trajectories.

Formal Boundaries

Bootcamp students universally reported wanting a full-time jobs in the software industry. Getting these jobs meant getting and passing job interviews. In their attempts cross these boundaries, four of our participants mentioned not understanding why they passed or failed interviews. This uncertainty was compounded by interviewers being unwilling to share their decision making process. As P5 said:

> [The problem is] not understanding what I'm doing wrong. [...] I would ask [interviewers], "Please give me feedback. What can I do better next time?" But I wouldn't get a response.

In spite of some uncertainty, bootcamp students mentioned several key factors in getting and passing job interviews.

The first was relevant educational credentials. While three of our participants had degrees in CS (bachelor's or associate's), the rest did not. Some of our participants chose to attend a bootcamp as a way of getting relevant educational credentials that would help with job interviews. P6 believed their bootcamp did just that, but several of our participants felt that bootcamp certificates were looked down on by employers. P3 said there was a “stigma” against bootcamp certificates and P26 explained:

> The fatal mistake I made was thinking that code school represented a vocational training standard, that it’s somehow equivalent to going to nursing school and getting a certificate that says, “I’m qualified to be an entry level nurse.” [...] It simply doesn’t work that way.

Second, bootcamp graduates talked about the need to get initial software industry work experience (six participants describe this with a version of the phrase “get a foot in the door”). To get initial experience, some bootcamp graduates found paid contracting work and internships. Six participants were in internships and three were in a bootcamp that included internships. We believe our data over-represents internships since several participants were recruited through others in the same internship and we heard little else about internships besides how most were not open to bootcamp graduates (P16 said, “A lot of the internships [...] only want college-aged computer science students.”).

Third, several of our participants mentioned the importance of online portfolios in getting a job. Some said their bootcamps gave them enough time, knowledge, and projects for their portfolio, while others used additional time and effort after graduation.

Fourth, in order to find job openings and meet recruiters, our participants talked about the need to network by going to tech meetups and hackathons, applying for jobs, and using LinkedIn and bootcamp Slack channels. P8 had a programming background, but chose to attend a bootcamp in part for the networking and P3 believed networking made a large difference in getting a job:

> At that point it didn’t really matter how good you are at coding. [...] Some people were always behind in their coding, but they got jobs straight away because they had the networking connections.

Fifth, our participants emphasized the importance of interviewing skills, especially the skill of “whiteboarding” (eight students used a variation of that word) an interview technique, often requiring knowledge of data structures and algorithms. Some students approved of the whiteboarding training at their bootcamps, while others felt they needed more practice than their bootcamp gave them. Students used a variety of methods to get more whiteboarding practice, from online courses, to whiteboarding practice meetups, to non-bootcamp in-person classes.

Besides whiteboarding skills, several of our participants mentioned soft skills interviewers were looking for. P10 mentioned needing to be “a cool person,” and P1 listed several specific factors:

> Soft skills are attractive to people right now that are hiring coders…I felt when I was in interviews they were saying that they want someone with strong communication skills and someone who’s easy to work with, a team player, who took instruction well.

Informal Boundaries

Our participants’ discussion of informal boundaries fell into three categories: *knowledge*, *identity*, and *belonging*. The *knowledge* expected of a software developer included “learning to learn,” meaning the ability to learn new programming languages, APIs and libraries from documentation, tutorials, and websites (like StackOverflow). Twelve students mentioned this concept. P7 said this type of learning is why they chose to attend a bootcamp in the first place:

> There’s lots of beginner stuff and lots of advanced stuff [online], but I needed the middle part of learning. […] So I decided to apply [to a bootcamp].

Most said this skill was something their bootcamp taught them, though some were not happy with how this was taught, such as P22 :

> So they’re trying to get you into this mentality of you have to read all the documen-

> tation. They sit back in the background [to let students read the documentation], and what annoys me is that I've paid a lot of money so that I could have somebody there to teach it to me.

P11 wished his bootcamp had focused more on developing this skill in JavaScript:

> [I wish my bootcamp had said:] "JavaScript is the Wild West. There's all of these different libraries. We're gonna teach you one to sort of show you what it's like to teach yourself a given library. Now go out and teach yourself whatever you need to for any given job."

Another piece of knowledge expected of software developers was knowing popular technologies and practices. P12 mentioned learning at their bootcamp about programming tools like Git and Slack, while P22 said they went to a bootcamp because they "wanted to learn the technologies that are up-and-coming."

The second category of informal boundaries was *identity*. Some students said they had difficulties in claiming an identity as a software developer and felt *impostor syndrome*.[6] Impostor syndrome was mentioned by seven of our participants (though one said they didn't struggle with it). One student (no CS degree) said that even after working as a software developer for about six months, they "still don't feel like an established developer." P2 said their bootcamp encouraged them to publicly claim the title of "web developers:"

> At the end of the first week they said, "Bring up your LinkedIn profiles [...] and change [your title] to web developer." And we're all like, "What? You've got to be kidding me. We're not web developers yet." And her point was that until you start thinking of yourself as one, then nobody else is going to.

The third category of informal boundaries was *belonging*, or fitting in among software developers. This included needing to know "the terms that interviewers are looking for," (P7) and handling "the social aspects that allow you to be a part of this group" (P5). One

[6] Impostor syndrome is when someone falsely believes that they are not competent and that they have fooled anyone who thinks they are [18].

of the most mentioned aspects of fitting in was the lack of women. One participant (male, CS degree) said his current work environment was “all white dudes.” Another participant (female, no CS degree) worried about how to handle a male-only work environment:

> In an actual job [when] I’m the only woman on the team, how do I do that?

Several participants said gender dynamics played into why they did not learn software development earlier. One participant (female, no CS degree) said:

> I never thought when I was younger that women could be programmers. That’s just something that everybody knew, I guess.

Another participant (female, no CS degree) said her bootcamp provided a second chance to become a software developer:

> I’m a good example of somebody who easily could’ve gotten into this field the first time around. When I was in college [...] it just wasn’t floated as something I could do. Nobody ever said, “Oh, you can’t be a computer scientist.” But nobody ever said, “Oh, you can be a computer scientist,” either.

Race also came up as an element of fitting in. One Black participant (female, no CS degree) mentioned the lack of diversity at her current company:

> Especially at [my company], I felt like a lot of the software engineers I had seen are white or Indian or Asian. I see very few women, I see very few black people, so it’s hard,

An Asian participant (male, no CS degree) said he was used to being a minority, but in the “programming world [he] wasn’t a minority anymore,” and a Latino participant (male, CS degree) said his race was less of an issue in Seattle and in the software industry than in his hometown:

> [In my hometown] it always felt like [...] I was just the brown guy. [...] Coming out here, people are a lot more open minded and they don’t care what you are, they just care what you’re doing.

Besides race and gender, the perception of software developers as nerdy or intelligent played a role in fitting in. P25 said:

> I started [learning to code] online. But it was so foreign to me that I'm like, "Okay, these are just for nerdy people. There's no way I'm gonna be able to."...I get this now but once it gets to the more difficult stuff I probably won't get it because it's probably only for these kind of people, and I don't think I'm that kind of person.

Similarly, P2 perceived back-end programmers as "really technical people who eat Linux for breakfast," and then was surprised to find they also enjoyed back-end programming.

For others, the perceived nerdiness and intelligence of software developers was a desirable feature. One (female, no CS degree) had negative experiences with the online gaming culture, and needed to "start learning if [the software industry] is a community I can stand." She said:

> When I actually went [to a tech meetup, the fact that I was new] wasn't any kind of barrier, [...] people were actually supportive. It was like, "Oh, have you tried this? Oh, it's fantastic. You should really check this out." Or,"If you're gonna go to this event, I'll tell you who you should talk to." They were really helpful.

P16 felt similarly about meeting software developers:

> You know, when I went to a lot of events before I started bootcamps, I thought, [...] "I feel like I fit in with the curiosity and, for the most part, level of intelligence," even though I didn't have any of [programming] skills yet. I was like, "I feel like this is a, for the most part, understanding and open community that's dedicated to a lifetime of learning," and I thought, "I can totally work with these people." So I hope that it works out eventually.

Bootcamps' Role in Negotiating Boundaries

Bootcamp students had to negotiate both formal and informal boundaries in the software industry, and our participants expressed different views on how attending bootcamps contributed to this process. Some participants believed their bootcamps were successful in

Table 2.1: Times spent before and after bootcamps

	Time spent before bootcamp	Time from graduation to job		Time from graduation to our interview (no job)	
	Classes (no CS degrees)	Full-time job	Internship	With some contracting	With no contracting
Time Range	3 months - 6 years	2 - 5 months	0 - 1.5 years	3 - 12 months	1 week - 9+ months
Median Time	9 months	2 months	1 week	5.5 months	2.5 months
Participant #s	9, 16, 20, 26	1, 2, 3, 6,	4, 5, 7, 8, 9	10, 11, 12, 13	15, 16, 17, 18, 26

getting them what they needed to know, such as P4:

> I would say that going to [my bootcamp] was probably the best decision that I've ever made [...]. Going from not knowing anything about coding [six months earlier] to being here today is pretty ridiculous...I love [my bootcamp].

Some were upset with their bootcamps. P12 said some from their cohort "want[ed] to do a class action lawsuit," and P26 said, "if I were able to do it all over again, I absolutely would not go."

Others had mixed feelings, such as P16, who said bootcamps and other classes were "just steps along the path [into the software industry] that every person has to find," and another participant (male, CS degree), who said:

> I feel very confident being able to get a job now. And I do attribute it to how things went while I was at [my bootcamp]. But [...] I withhold some judgment on how good [my bootcamp] is at producing employment for people on a broader scale.

Several students were skeptical of the success rates their bootcamps advertised. One (female, CS degree) questioned whether contracting work was being counted as success and P11 said their contracting work was announced as successful employment. One bootcamp had a (later discontinued) job guarantee program with strict requirements which one student (male, no CS degree) missed at the end. Another student thought these strict requirements were used to make the bootcamp success rate look better.

Personal Obstacles

Bootcamp students also faced *personal obstacles* in entering the software industry. Most of these personal obstacles stemmed from the time it took to transition into the software industry. While a number of students told us they chose bootcamps because they provided a faster route to a job than a degree, many still found time to be an obstacle. Students could spend a year or more when including the time spent learning programming before their bootcamp, or learning more and seeking full-time jobs after graduation (Table 2.1)[7]. Many felt their bootcamps had not communicated accurately about the time needed. P16 said, "I wish I had known before I started [the bootcamp] that it could take a really long time." In addition to losing time in the career change process, the career change also could mean loss of previous career and educational investments, like P9 said:

> I knew if I went into coding, I would be making my bachelor's degree obsolete. And that was a hard thing to cut my losses. But I just had to realize it's not what I wanna do. So it's not worth it to me to keep doing something I don't wanna do.

Financial costs were a personal obstacle for students trying to enter the software industry. These costs came from bootcamp tuition and prolonged unemployment. P1 said, "the cost became more of an obstacle after graduation, when I was on the job search," and another participant (male, CS degree) said he was surprised by this cost:

> When I [started the bootcamp, I] was really surprised [that after] almost four months [...] there was a decent number of [the previous cohort] still not having jobs. [...] I certainly hadn't factored that into my finances.

These financial obstacles were mitigated in various ways. Some participants had a spouse who supported them. One participant went back to part-time work after their bootcamp while continuing to study on their own. Bootcamps sometimes offered partial solutions through paid TA positions that graduates could take while they were job-hunting. Some

[7] Our participants did not always tell us how much time they spent on different activities, particularly with activities before bootcamps.

students still struggled tremendously. P26, who could not find a job after attending, said:

> I have been so distressed by [the bootcamp and job search]. I have put everything on hold. My house is for sale. My whole life is in shambles because of this. The whole thing has pretty much derailed my career, derailed my life. I spent tens of thousands of dollars pursuing this.

Another participant (female, no CS degree) said:

> To be extremely honest, in choosing this path, I've come the closest to being homeless that I've ever been.

The obstacles of time and money could be compounded for those with families. One participant told us about the nine months following bootcamp graduation:

> I pretty much devoted my time to [my bootcamp's] prescribed job hunting methods, which means financially, I have no money. I can't go work because I really needed to do job searching full time. And that [sacrifice] reflects on my family because now we're low on funds [...] and now instead of selling our house and buying a house, we're selling our house to pay the debt that we're in and then go rent until I can find a job.

In addition to time and money were several other obstacles. Finding support of family and friends was an obstacle for some, including one participant (female, no CS degree) who said:

> My friends and family [...] have known me until that point as nonprofit lady who did informal education and experiential education. So when I said, "I'm doing this program so I can be a software developer," they'd just look at me like, [...] "You doing tech? We just don't get it."

Location was another. Though some said the software industry opened opportunities to live where they wanted, others had to leave friends and family. As one participant (female, no CS degree) said:

> I love [my state]. I have a house there. And my husband's currently there. I wanna

> go back, but at the same time, [my internship in Washington is at] one of the top tech companies in the United States.

Another set of obstacles involved motivation. Several students mentioned the difficulty of maintaining focus while learning software development outside of a bootcamp. For example, P10 said:

> I didn't want to commit towards something that I wasn't passionate about, and regular school is boring. [...] I needed to go to a bootcamp, because it's going to keep me focused.

Motivation to persevere on the job hunt was an obstacle for some, like P5, who said they applied to 100 different jobs and P1, who described job hunting as "dehumanizing."

Finally, confidence was an obstacle for some students, which was previously discussed in terms of impostor syndrome in 2.4.2. For some students, attending a bootcamp increased their confidence, (P1 said "[My bootcamp] made me very confident about my ability to achieve the goals that I've set for myself as long as I work hard."), while others lost confidence in a bootcamp (P18 said, "My confidence went downhill after that month at [the bootcamp].").

2.4.3 Barriers in Coding Bootcamps

We now turn from the barriers students faced in entering the software industry to those they faced specifically in bootcamps.

Formal Boundaries

Formal boundaries in bootcamps included admissions, payment, co-location, and bootcamp stages. Admission to a bootcamp could be permissive (one student said their bootcamp had "no entrance exam or anything [...] they'll take literally anyone.") or strict, such as the first one P10 applied to:

> It's super competitive. The acceptance rate I think is 2% [...] I didn't get in, which

> is fine. So that's why I went to [another bootcamp].

After admission, all bootcamps required in-person attendance (at least for some sections of the bootcamp) and significant payment. One bootcamp had no tuition, but for the others, our participants mentioned prices from $10,000 to $20,000. Some bootcamps offered scholarships and some allowed students to pay partial tuition for only attending part of the bootcamp. During the bootcamp, courses or stages were formal boundaries marking progress. Some bootcamps had tests that had to be passed in order to advance. When students graduated, they could stay involved through alumni networks, residencies, and TA positions.

Informal Boundaries

Informal boundaries within bootcamps were often similar to those in the software industry (2.4.2), particularly those of race, gender, expectations around knowledge, impostor syndrome and the perceived "nerdiness" and "intelligence" of software developers. For example, the demographic makeup of many bootcamps had a lack of women and minorities like the software industry. P24 said that there were only two women in their cohort, and another (male, CS degree) described the ways his cohort was homogeneous:

> Almost everyone was in a really tight age band. It was a bunch of people that were 27 years old. Everybody was white. Everybody was middle-class, wealthy though there were a couple outliers. [...] The only way that [my bootcamp] was diversifying at all from the current demographic of people in software was there were a lot of women there.

On the other hand, some bootcamps pushed for more diversity. For example, one bootcamp only accepted women and people of non-binary gender, and at least two bootcamps had built-in training around diversity and empathy. One (female, no CS degree) explained how welcoming she felt her bootcamp cohort was:

> There are [many] of us that come from poor backgrounds. There are a number of

> us that are Latina. [...] [My bootcamp] is the first place where I felt that owning different identities and being different is okay.

A different kind of diversity at bootcamps was in students' relations to programmers. Though we did not specifically ask, we found that at least four students were married to programmers, and another seven had parents, siblings or friends who were programmers. One student (male, CS degree) said:

> Yes, [there are] women being involved in programming, but the women the bootcamps are drawing in right now are from the same social sphere as the current programmers. [...] In fact, it's a lot of spouses of programmers.

As with diversity, the informal boundaries around perceptions of "nerdiness" and "intelligence" showed up in bootcamps. For example, P22 said they had difficulty relating to classmates who were gamers. Similarly, students mentioned feeling impostor syndrome in their bootcamps. In particular, several students mentioned their cohorts being divide into two groups. There were different descriptions of the divide based on one or more factors including effort, "being good at school" (P16), being "tech savvy" (P3), and seven people mentioned a divide based on "background" and previous "experience" with programming, such as P18:

> It was divided, the class. Those with experience, I think, they were looking down at [those of us without experience] because maybe there were certain things we were supposed to know and we didn't. So when we had to work on common assignments or in pairs, it was a bit of a barrier.

Another participant (female) saw this divide from the other side:

> A lot of [the other students] don't have the experience that I have. I have a degree in computer science, I have 10 years-plus experience in a job market. [...] A lot of people are coming from accounting, or something else completely unrelated [and] are probably are going to have a way harder time than I am.

This divide was difficult for some students. For example, P25 talked about a student who was having trouble and then quit:

> To me what was most sad was not the fact that he quit, [but that] he felt he was dumb and not smart enough to do it.

To cope with this divide, some students tried to reach out within their cohort, like P12 who hung out with more experienced programmers, even though they did not feel like they fit in with them.

Though some students talked about divided classrooms, other students mentioned group bonds that formed in their bootcamp. P5 mentioned making close friends at their bootcamp and P9 said at their bootcamp, “everybody knows what’s going on with everybody else. It was a very close-knit experience.”

One final informal boundary faced by students was access to teachers. While some participants at some bootcamps said their teachers were helpful and engaged with everyone, other participants felt differently, such as one (female, no CS degree), who said “I felt uncomfortable asking questions [of the teachers].” One (no CS degree) had a particularly bad experience with asking for help:

> There was this one time where my database wouldn’t work because I hadn’t capitalized a letter and I asked one of the assistant teachers about that and he thought it was ridiculous that I made a mistake about this capital letter.

Some participants saw bias in who their teachers spent time with. One participant (male, no CS degree) believed some women were getting extra attention and another (female, no CS degree) said extra help was “reserved for people who were on the upper-end of class.” TAs provided a middle ground of access between students and teachers, though opinions ranged from, “It’s very nice that we have TAs” (female, no CS degree), to “The TAs were not helpful whatsoever” (female, no CS degree).

Personal Obstacles

Many of the personal obstacles faced by bootcamp students in their software industry trajectories (2.4.2) overlap with those they faced in attending and succeeding in bootcamps, such as time, money, impostor syndrome, and location. For example, just as location was an obstacle for some jobs, one participant (female, no CS degree) moved away from her husband to attend a bootcamp.

The ways personal obstacles were unique in bootcamps revolved around what eight students described as the "intensity" of the bootcamps. The intensity started with a large percentage of students' weekly time spent on the full-time portion of their bootcamps.[8] One participant (female, no CS degree) said that the official weekly schedule of her bootcamp was eleven hour days, six days a week, while P18 talked of even longer days:

> Ten, twelve hours at least per day, and sometimes fourteen or sixteen hours [...] and no weekends because we had assignments.

This time spent gave our participants very little time to do other things in their life. One (female, no CS degree) talked about the resulting state of her home and hygiene:

> I did all my laundry this weekend, for the first time in like a month, because I was out of everything. My kitchen is a disaster. My whole house is just a mess. Anything that is not directly related to [the bootcamp] or to keeping me up and functioning, just goes by the wayside. [...] I don't remember the last time I had a shower.

The time spent at bootcamps added financial obstacles beyond just tuition and costs of living. Students were not able to hold jobs for full-time portion of their bootcamps and P9 said financial difficulties caused some people to drop out of their bootcamps.

The intense time commitment of bootcamps also meant students lost time with friends (P11: "I had to tell pretty much everyone in my immediate intimate circle, 'I'm probably going to disappear.' "), partners (P21: "My poor boyfriend. I see him so rarely.") and

[8] Some bootcamps were broken into stages and they allowed or required the early stages to be taken online or as night classes.

family (P12: “I didn’t spend time with my family at all for a month.”). Also, similar to the software industry trajectories 2.4.2, some students faced obstacles in getting social support for attending bootcamps.

The intensity and speed at which material had to be learned at bootcamps could be very stressful for students. P11 said everyone else in their cohort broke down and cried at some point and one participant (female, no CS degree) said how this affected her “brain power:”

> Sometimes I’m just so burnt out, I can’t even think. I can’t process. Somebody asked me, “What’d you have for dinner last night?” I’m like, “I don’t know. I dumped all that.” I can tell you how to do this really cool loop and rails, but I have no idea what I ate. I just...I don’t know. It’s exhausting.

To succeed in the intense bootcamp environment, several participants said students needed confidence, commitment, and determination. P2 said, “What’s going to make or break your success [in a bootcamp] is how nice you are to yourself when you’re frustrated.” Another (female, no CS degree) got help from her husband:

> They say [their teaching style] is revolutionary [...] But for me, it didn’t work. [...] I learned more from coming home and my husband teaching me algorithms and how to approach a certain problem than [from the] teaching in class.

The intensity of bootcamps also had an effect on some students’ health. Two students mentioned how their diet had suffered (E.g., P5: “When I first started doing this, I didn’t really eat or drink too much.”) and three participants mentioned their lack of sleep while attending a bootcamp. P12 talked about getting sick:

> When I was in college, when I got sick, I could take some time off. At work, I got sick and they’d rather me stay home. Here, when I got sick, I needed to still show up because one day of missing a class is a lot.

2.5 Discussion

Our study was the first peer reviewed research on coding bootcamps (Unlike prior reports on bootcamps [99, 117, 100]), and as far as we know is currently the only published research to explore the experiences and perspectives of bootcamp students (Burke et. al. later reported on the perspectives of employers and instructors, not students [15], though we know of more work that focuses on student perspectives that has not yet been published).

Our investigation provided a long, chronological perspective of several adults' attempts to enter the software industry (Fig. 2.2), and showed how bootcamps provided a second chance. In particular, some women, as in previous research [71, 11, 17], either had not thought programming was something for them or had been scared off by the lack of women in CS. The bootcamps provided focus, peers, networking, and a set curriculum for students, which helped them overcome knowledge barriers such as knowing how to seek for jobs, knowing general programming, algorithms and data structures (sometimes), the latest technologies, knowing the terminology and communication patterns used in the software industry, and team dynamics. Still, when trying to become professional programmers through bootcamps, many of our participants, perhaps due to their independence and experience or because of misalignments between bootcamps and the software industry, made use of additional time and resources outside of their bootcamps or even attended sections of multiple bootcamps. For these students, bootcamps were just one step on a longer path to cross the formal and informal boundaries into the software industry.

The informal community boundaries bootcamp students faced mirror prior work on computing education in high schools, colleges, and universities, such as those around race, gender, and previous experiences [71, 70, 11, 144]. Some, though, found different bootcamps (or cohorts) to be more open and inclusive. This could partially explain how coding bootcamps have achieved higher percentages of women [101] and may provide insights on how other computing programs can increase diverse engagement. Stereotypes of "nerdiness" and "intelligence" also formed informal boundaries for bootcamp students, as found elsewhere

[17, 11, 65]. The divide of classes into two groups, largely attributed to previous experience and prior knowledge, also matched other contexts [71, 70, 116].

Bootcamp students faced significant personal costs when attending bootcamps and changing careers. Some costs, like financial and family concerns, match what has been found in other career change contexts [103, 85]. Beyond those, the intensity of bootcamps and the career change time required significant perseverance and confidence, while leaving little time for relationships and self-care.

Though bootcamps offered more diverse graduates to the software industry, it was these diverse students who were taking on large costs and risks with few guarantees. Only one bootcamp had tuition covered by the industry, and several students doubted the success rates posted by their bootcamps. Additionally, students struggled with finishing their bootcamps, learning the material, knowing what was required to get a job, and a perceived "stigma" against coding bootcamp graduates. Some of our participants found full-time work despite these struggles (many were enthusiastic about their bootcamps), while others struggled or failed. These risks and costs may limit the diversity in background and financial status of those who attempt and succeed in entering the software industry through bootcamps. If coding bootcamps address the difficulties faced by their students and the industry takes on more of the risks and costs, then bootcamps have the potential to expand the pipeline into the software industry with more diverse talent, while personally benefiting many more students to come.

2.6 Limitations and Future Work

Our research focused on a small sample of students in coding bootcamps in one part of only one country. Other students may have had different experiences, especially in other bootcamps, in other places, and in other times. Additionally, while our stratified snowball sampling provided a range of experiences, we can make few claims about the commonality of experiences or causality.

Our interviews were also limited. We did not ask for a full chronology of events, which may have left gaps in the learning trajectories, and students may have answered differently with a different interviewer (all interviews were done by a white male with a CS background). More perspectives would give further context on bootcamps, such as from classroom observations and the views of bootcamp organizers and teachers, and those making hiring decisions in the software industry. Additionally, our use of prior frameworks in analyzing results could distort student perspectives.

To further understand the role of bootcamps in meeting demand for software developers, our results suggest future studies in the quality and content of instruction, the structural inequities within bootcamps and the software industry, and the downstream differences in long-term careers between software developers with CS degrees and with bootcamp training.

Chapter 3

PROJECT 2: A THEORY OF ROBUST API KNOWLEDGE

3.1 Introduction

As industry continues to expand, developers have created numerous application programming interfaces (APIs), including libraries, frameworks, web services, and other reusable code. For example, in the popular Node Package Manager (NPM), there are more than a half a million APIs available [2]. Students learn APIs in classes or in coding bootcamps (see last Chapter); developers learn and use them to build software in their jobs [9]. These APIs are what allow developers to make sophisticated software quickly by reusing the work others have done and shared, and developers consider available APIs as the most important factor when choosing a programming language [75].

When developers set out to use and learn APIs, they run they often find a gap between what they want to do with an API and what resources they can find. For example: official documentation is often incomplete, inadequate, or hard to navigate [110, 106, 136, 119, 31], relevant examples are difficult to find and to adapt [146, 119, 106, 30], crowdsourced documentation like Stack Overflow is often incomplete, incorrect, or misleading [92]. Without a deep knowledge of the API, they are likely to produce defective or brittle code [119, 60].

Though prior work has enumerated many struggles developers face in understanding API documentation, there has been less research on the underlying knowledge that documentation is trying to convey. Moreover, no prior work has defined an overarching theory of how and why this knowledge is necessary to learn an API. In this paper, we propose such a theory, defining a notion of *robust API knowledge* in the context of a developer in isolation understanding code that uses an API, considering options for using an API, and writing or modifying code that uses an API to achieve a specific behavior (though not even necessarily running the code).[1] We organize this knowledge around three components of knowledge that come together to form robust API knowledge: *Domain Concepts*, which exist outside of the API along with terminology; *API Usage Patterns*, which demonstrate how to coordinate the use of API features and show what is possible with an API, along with rationale for their construction; and *Execution Facts*, which allow a developer to predict the output and side effects of API calls given the different possible inputs. We contrast *robust* knowledge with *brittle* API knowledge, where a developer fails to understand aspects of an API, resulting in difficulty modifying, fixing, or using APIs successfully.

To provide initial empirical evidence for our theory, we ran two studies. The first study tested the role knowledge plays in understanding and task progress. We recruited students, created sets of tasks using four APIs, and controlled which types of core API knowledge students could access. We measured their progress and perceived understanding of code. Our second study tested whether the content of StackOverflow fit into our three core API types and how that information was perceived by users in StackOverflow. In the rest of this paper, we describe our theory, study, and results, then discuss their implications on API learning and the design of API instruction. This work is currently under review [131].

3.2 Literature Review

3.2.1 What is a theory?

There are many conflicting views on what constitutes a scientific theory and how to evaluate these theories [128, 19, 26, 52, 23]. The criteria that can be used to evaluate a theory include: how well it unifies and organizes facts and other components into a coherent model, what it captures about the meaning and internal logic of a situation, what it captures about causes and effect, what situations it applies to, how well it aids in answering questions in the field,

[1] In developing our theory, we chose to take a cognitivist perspective [22] rather than a sociocultural one. Though this ignores important factors like identity factors, social factors, communication factors, and developmental factors, it does allow us to focus on the content of knowledge about an API and build upon prior work with a similar focus.

what further questions the theory suggests, how well it fits current findings, whether the theory makes falsifiable hypotheses that can be tested, and whether those hypotheses hold up under experimentation [19, 26, 52, 64]. Which criteria a theory should be evaluated on depends on the particular purpose and use of the theory [19]. For example, falsifiability may be at odds with the purpose and use of some theories [19], but for theories that do produce falsifiable claims, an accumulation of contrary evidence may lead scientist to explore new avenues, potentially modifying the theory, or abandoning it entirely [52].

For this paper, we focus on the criteria of: unifying prior findings into a coherent model, capturing meaning and internal logic, and producing falsifiable claims. Thus, a successful theory of API knowledge would:

- Unify prior findings on API knowledge into a coherent model.
- Capture the internal logic of how parts of API knowledge relate to each other and the role API knowledge plays in helping a programmer work with code that involves APIs.
- Produces falsifiable claims about API learning that can be further tested.

3.2.2 API knowledge

Prior work on APIs has investigated the challenges and process of using and learning APIs. Developers seek knowledge from tutorials, documentation, experts, or in-line comments in code [68, 91, 57, 43]. Developers use different strategies in learning APIs (e.g., trying to understand everything before touching code or modifying unknown code to see what it does) [21, 20, 73]. Most developers prefer the strategy of looking at examples [73, 106, 59, 126, 30, 110, 115, 88, 140, 118, 146].

When when developers try these strategies, they often find a gap between what they want to do with an API and what resources they can find. There are gaps in formal [110, 106, 136, 119, 31] and informal [92] documentation. When seeking examples, developers can sometimes find examples of how to use an API that they can copy and paste into their code without needing to understand the API [119]. When an example exactly matches their needs, this method can be effective at helping developers leverage an API [13]. However, this method does not always work. If a developer cannot find an example that is exactly what they need, or they cannot determine whether a found example is relevant, or they need to modify their code later, they will need a deeper understanding to successfully adapt it. Without that knowledge, they are likely to produce defective or brittle code [119, 60].

Though previous work has not offered an overarching explanatory theory of API knowledge (at least by our definition in 3.2.1), there have been many discoveries that pave the way toward a theory. These studies have found that developers need to know:

- What possibilities an API allows a developer to create [73]
- Precisely what behavior they are trying to achieve [59]
- How to set up an environment in which they can use the API [73]
- An overview of the API's architecture, including its components and structure [73]
- The purpose of code they want to reuse and the rationale for its pieces [57, 106, 6]
- How to search for and recognize relevant information; this involves knowing search terms and concepts [31, 58, 59, 146, 53]
- How the abstractions of an API map to abstractions used in the domain of the API [20, 62, 47, 6]
- How to use API features (e.g., object construction) [30, 59, 81, 6]
- Dependencies and relations between parts of APIs or different APIs [31, 30, 59, 81, 29, 53]
- Common or useful usage patterns and best practices [110, 146, 47, 53]

- Understanding the relationship between code and output [59, 57]
- How to find run-time information and debug programs [59]
- Runtime contracts, side effects, return values, and other properties of API behavior [43, 62, 29]

Additionally, researchers focusing on general software engineering have emphasized:

- *Programming plans*, which "represent stereotypic action sequences in programming" such as the use of algorithms or data structures [122]
- *Domain plans*, which are sequences of actions considered at the domain level, separate from the code and low-level algorithms. [139]

3.3 Theory of Robust API Knowledge

3.3.1 Three Components of Robust API Knowledge

Building upon prior work, our own experiences learning APIs, observing developers use APIs across many studies, and teaching APIs in classes, we created a theory of robust API knowledge that distills all prior evidence into a simple, explanatory theory of what API knowledge is and how it structures API learning. Our theory begins from the premise that *using* an API, much like any programming, involves imagining requirements for program behavior, and then searching for a program that meets those requirements. This process requires learning three classes of knowledge (Table 3.1) : *domain concepts* that exist in the world outside the API which the API attempts to model, along with the terminology used by the API; *execution facts* about the API's runtime behavior that summarize that behavior into rules about inputs, outputs, and side-effects; and *API usage patterns*, which are modifiable code patterns, along with rationale for why the pattern works and is organized as it is;. We argue that these three components of knowledge, when present, allow a developer to

Knowledge Component	**Definition**	**Understanding of code due to knowledge component**	**View of the design space due to knowledge component**
Domain *Concepts*	Abstract concepts that exist apart from the API and the terminology used by the API and documentation	The conceptual purpose of the overall code and each of its components	Theoretical solutions in domain (may or may not be supported by the API)
Execution *Facts*	Facts about input, output, and side effects of API calls and references given the different possible inputs	Predict what each API call or reference will do	Potential API calls and references to the API
API Usage *Patterns*	Patterns of code used by the API and rationale for the patterns in terms of *concepts* and execution *facts*	The organizations of code using an API: how pieces of code can and do relate to each other	Potential programs using API

Table 3.1: Summary of the three components of robust API knowledge.

productively find relevant abstractions in an API and correctly leverage the API's behavior to compose API abstractions into programs that meet their goals. We therefore refer to the set of these three components of knowledge as *robust* API knowledge.

In the rest of this section, we describe and justify these components of API knowledge in more detail, then return to discuss how they interact, what other knowledge is required but outside the scope of our theory, and how our theory meets criteria for falsifiability, explanation, prediction, and consistency with prior evidence.

Domain Concepts

Domain concepts are abstract ideas that exist outside of an API, which an API attempts to model, and the specific terminology that the API and documentation use to refer to the concept (see Fig. 3.1). These concepts are any idea modeled in software. For example, this paper is written in LaTeX, which models concepts from the domains of typography and technical writing like fonts, glyphs, tables, and cross-references. These concepts exist in the world, outside of LaTeX, and are modeled in LaTeX. Our theory argues that someone learning the LaTeX APIs needs to understand both the concepts that LaTeX models from

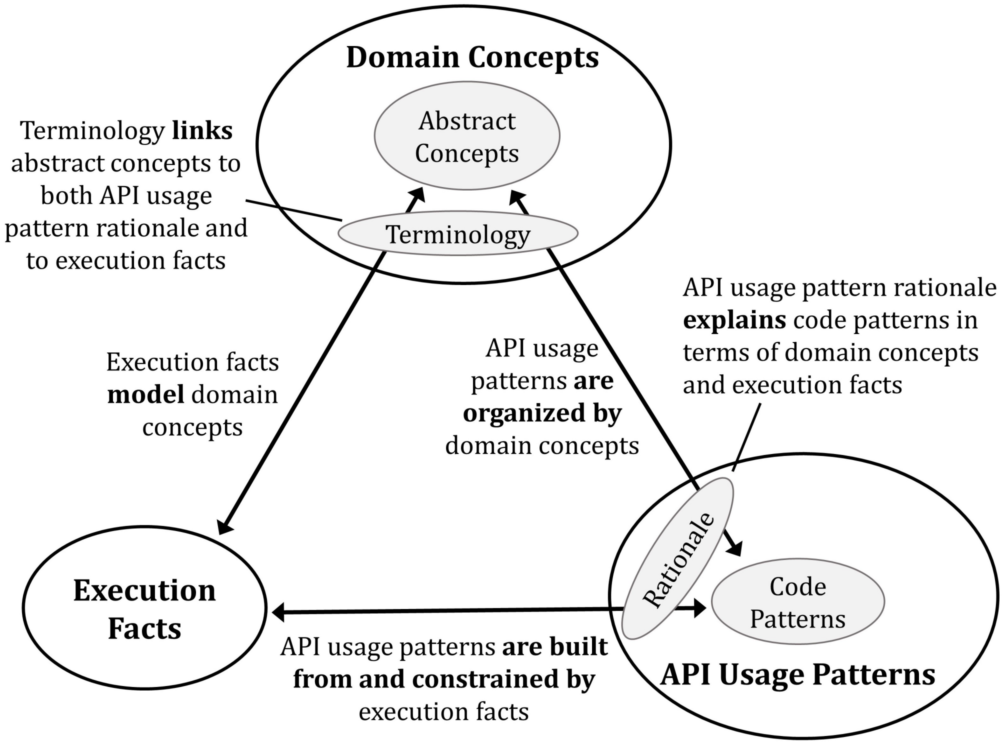

Domain Concepts
Abstract Concepts
Terminology
Terminology **links** abstract concepts to both API usage pattern rationale and to execution facts
Execution facts **model** domain concepts
API usage patterns **are organized by** domain concepts
API usage pattern rationale **explains** code patterns in terms of domain concepts and execution facts
Execution Facts
Rationale
Code Patterns
API Usage Patterns
API usage patterns **are built from and constrained by** execution facts

Figure 3.1: Relationships between the three components of API knowledge.

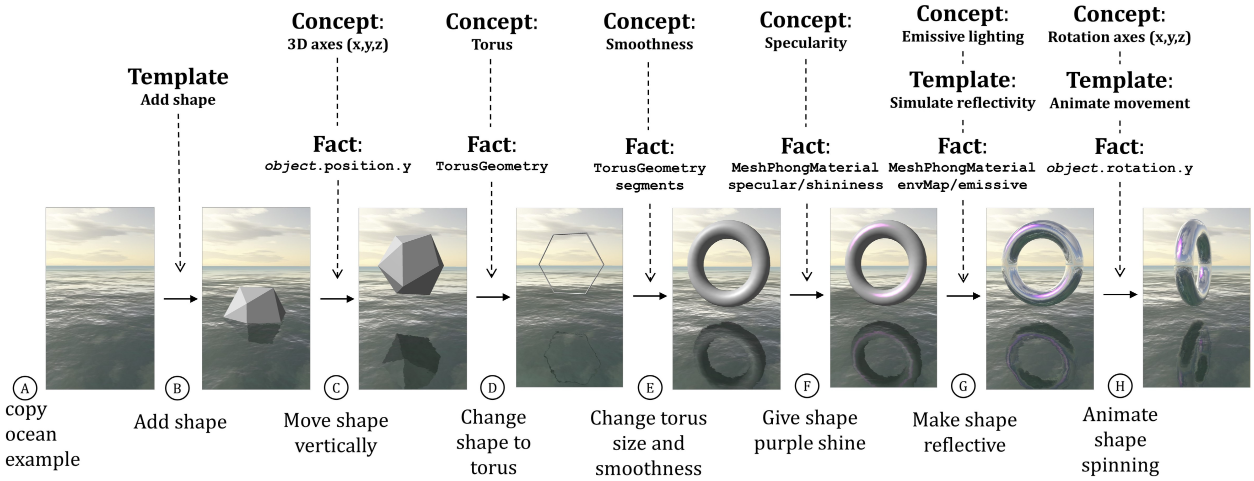
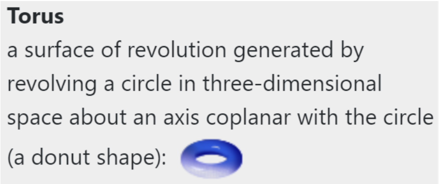

Concept: 3D axes (x,y,z)
Concept: Torus
Concept: Smoothness
Concept: Specularity
Concept: Emissive lighting
Concept: Rotation axes (x,y,z)
Template Add shape
Template: Simulate reflectivity
Template: Animate movement
Fact: *object*.position.y
Fact: TorusGeometry
Fact: TorusGeometry segments
Fact: MeshPhongMaterial specular/shininess
Fact: MeshPhongMaterial envMap/emissive
Fact: *object*.rotation.y
A copy ocean example
B Add shape
C Move shape vertically
D Change shape to torus
E Change torus size and smoothness
F Give shape purple shine
G Make shape reflective
H Animate shape spinning

Figure 3.2: The stages a developer might go through to create a spinning 3D ring floating over an ocean using the ThreeJS API. We propose knowledge needed to perform each step, which is further explained in section 3.3.1. In study 1, this was one of the four task sets given to participants.

typography and technical writing, but also how those concepts are referred to within LaTeX. For example, to understand what the command `textit{}` does, one needs to understand the concept of italics, and that the "`it`" in `textit{}` refers to the concept of italics. APIs might include concepts from multiple domains, so an API might calculate screen layouts (UI design domain) by using constraint solvers (computing domain).



Torus

a surface of revolution generated by revolving a circle in three-dimensional space about an axis coplanar with the circle (a donut shape):

Figure 3.3: Concept definition of a Torus. Used in study 1.

Consider another example, this time from an API called *ThreeJS* (which we later use in Study 1), which is used to create 3D animations in web browsers. Figure 3.2 shows several iterations of a loading screen a developer might want to create, consisting of a reflective 3D ring spinning above an ocean. In this imagined design exploration scenario, a developer must have knowledge of numerous concepts to succeed at each step. For example:

- To move a 3D shape vertically (stage C), a developer needs to understand concepts of *3D axes* and the names of these axes in ThreeJS (x, y, and z).

- To make a ring (stage D), a developer needs to know that the ring shape they are seeking is called a *Torus*. Figure 3.3 shows how the concept of a torus might be explained.

- To give their shape a purple shine (stage F), a developer needs to know that this is called a *specular highlight*, how specular highlights behave in 3D scenes with lighting, and how ThreeJS refers to specular highlights.

- To make the shape reflective (stage G), the developer need to know that *emissive lighting* makes a shape a uniform color unaffected by light (so that later they can make the reflection and the specular highlight the sole determinants of the shape's color).

- To make their shape spin (stage H), they need to know what *rotation axes* are, how ThreeJS refers to them, and, in particular, that rotation around the y axis is their goal.

Broadly, our theory argues that domain concepts help developers in a number of specific ways. Concepts help developers consider what may be possible in an API, manipulate API abstractions that align with those conceptual abstractions (as mentioned in prior work [20]), and help a developer understand the purposes of API abstractions and code using the API (see Table 3.1). Additionally, knowing the concepts and terminology helps developers find and recognize relevant information about an API, both of which prior work has shown are critical to API learning [73, 58, 146, 30, 31, 53].

One implication of the claims above is that more concepts related to an API a developer understands, the more flexible and expressive they can be in working with it. For example, while our ThreeJS task did not demand the use of concepts from 3D graphics like textures, particles, or skeletons, knowledge of these may increase what a developer can conceive of trying in ThreeJS.

While our theory claims that domain knowledge is essential to robust use of an API, it makes no claims about how well a developer needs to know these concepts. A coarse, but mostly correct notion of fonts may be sufficient for using LaTeX, but a more nuanced concept of font faces and families may give a developer greater control over the precise typographic rendering of text. A muddled idea 3D scene lighting might be sufficient for some tasks, but limit a developer's expressive range with ThreeJS. Each API, and each behavior one might try to implement with an API, is likely to have its own knowledge requirements.

Execution Facts

Execution facts are declarative knowledge in the form of simplified rules about an API's underlying execution behavior, sufficient for predicting, understanding, and explaining API execution. Each of these facts models some set of concepts in terms of programming constructs such as types, inputs, outputs, and side effects of executing parts of an API (see Fig. 3.1). These facts can be at different levels of abstraction, from low-level information like the effect of function arguments on function return values, to higher-level information about the APIs internal state and the control and data flow of an API's global behavior [62]. Execution facts also extend to failures to model concepts as the API claims to (implementation bugs) and the dependence of API code's behavior on the environment (e.g., 'this code will not work in Internet Explorer').

For example, completing our ThreeJS scenario (Figure 3.2) might require knowing these facts:

- To move an object vertically (stage C), a developer needs to know that the property `*object*.position.y` changes the object position.

- To create a Torus object (stage D), a developer needs to know that the `TorusGeometry` constructor creates a Torus which can be rendered when added to a scene.

- To create a smoothed Torus with the target size (stage E), a developer needs to know facts about the effect of the first four parameters of `TorusGeometry` (Figure 3.4). In particular, they need to know that the first two (`radius` and `tube`) define sizes and the second two (`radialSegments` and `tubularSegments`) define geometric smoothness.

- To give the object a purple shine (stage F), a developer needs to know the behavior of options for MeshPhongMaterial, in particular that `shininess` defines how sharp a specular highlight is and `specular` defines the color of the specular highlight.

- To make an object appear reflective while maintaining the purple specular highlight (stage G), a developer needs to know that `MeshBasicMaterial` does not allow specular highlights and that they must use `MeshPhongMaterial` instead, setting the `emissive` color to white to make it uniformly lit and turning off the main `color` by setting it to black. They still need to leave the `specular` and `shininess` values to keep the purple shine.

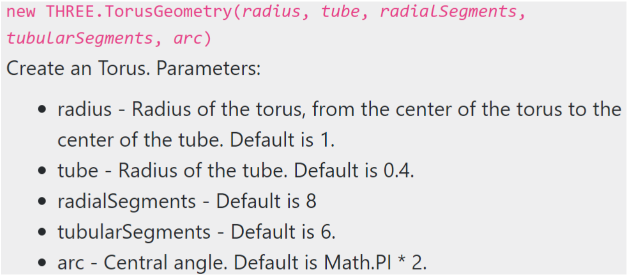
- To animate the object (stage H), the developer needs to know that the property *`object`*`.rotation.y` changes an object's rotation.

- For all steps they need to know that changes will be visible on the next frame rendered.

As can be seen in the list above, these facts that a developer needs to know model concepts (either external concepts or API specific concepts) in particular ways. For example, one way the concept of smoothness is modeled by the API is by approximating smoothness with geometric segments (`radialSegments` and `tubularSegments`). Our theory claims that developers need to know how the API specifically *models* domain concepts as execution facts.

Any non-trivial API may have innumerable facts that one could learn; to implement a particular behavior, however, there might only be a small set of facts needed to be able to accurately reason about and predict program behavior.

The ability to reason about and predict API execution behavior is central to many software engineering activities. It helps developers efficiently test, debug and repair code [110, 59], interpreting error messages they receive, or other unexpected behavior. Additionally being able to predict API behavior helps developers judge between options when modifying code or writing new code.



```
new THREE.TorusGeometry(radius, tube, radialSegments,
tubularSegments, arc)
```

Create an Torus. Parameters:

- radius - Radius of the torus, from the center of the torus to the center of the tube. Default is 1.
- tube - Radius of the tube. Default is 0.4.
- radialSegments - Default is 8
- tubularSegments - Default is 6.
- arc - Central angle. Default is Math.PI * 2.

Figure 3.4: Fact for the TorusGeometry function in Threejs. Note that by default, tubularSegments is 6, making the default shape a hexagon. Used in study 1.

API Usage Patterns

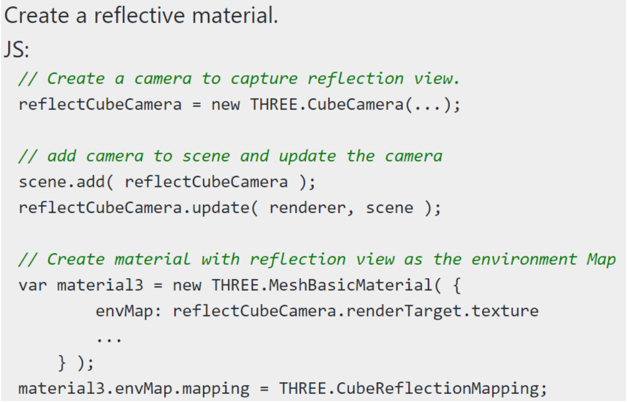
API usage patterns are some form of a code pattern (e.g., code examples, ordered lists of API calls) that conveys how parts of it may be modified. Given how often multiple APIs are used together, these patterns might include the use of multiple APIs in coordination. We additionally consider API usage patterns to include rationale (whether explicit or implicit) for pattern's construction in terms of both the concepts that the code is organized around (e.g., the code may implement a specific algorithm, heuristic, trick, or convention known in the domain) as well as how the execution facts of the API work together (or must be worked around) to produce the desired result (see Fig. 3.1). [2]

For example, consider the JavaScript Canvas APIs, which renders 2D graphics in an element of a web page. One common task is rendering a rectangle:

```
ctx.rect(20,20,150,100);
```

This code snippet is not generalizable, nor does it show any coordination in the use of the API (it isn't even executable). To show coordination in use of the API (and make it

[2] Various definitions of API "usage patterns" have been used before (see the Robillard et. al.'s literature review [107]). These have generally been defined concretely in terms of ordered or unordered sets of API calls, and sometimes with additional information like control structures [87]. Our definition here is more general, and unlike those definitions, we include rationale.



Create a reflective material.

JS:

```
// Create a camera to capture reflection view.
reflectCubeCamera = new THREE.CubeCamera(...);

// add camera to scene and update the camera
scene.add( reflectCubeCamera );
reflectCubeCamera.update( renderer, scene );

// Create material with reflection view as the environment Map
var material3 = new THREE.MeshBasicMaterial( {
        envMap: reflectCubeCamera.renderTarget.texture
        ...
    } );
material3.envMap.mapping = THREE.CubeReflectionMapping;
```

Figure 3.5: API usage pattern (in template form) for creating a reflective material in Threejs. Used in study 1.

more executable), a more extended example might be:

```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.rect(20,20,150,100);
ctx.stroke();
```

This demonstrates coordinated use of the canvas element, the drawing context for that element, the specification of a rectangle path, and the rendering of that path. However, it does not explain any of these parts nor why they are necessary nor how it might be modified. An API usage pattern that does all these things might look like this:

```
// To draw anything, first get the Canvas element in which to draw. It doesn't matter how you get it.
var c = document.getElementById("myCanvas");
// Each Canvas element has an object that represents a drawing context, where all drawing operations happen.
// "2d" supports two-dimensional drawing. Others options include "webgl" and "webgl2", for 3D rendering.
var ctx = c.getContext("2d");
// With a context, we can draw a rectangle. This function, however, only specifies a path for drawing.
// You must call a stroke() or fill() command before the browser renders anything. All positions,
// widths, and heights must be within the boundaries of the Canvas's coordinate system to appear.
ctx.rect(20,20,150,100);
// Now, we can apply a stroke to the path. We could have instead called fill() to fill the path with color.
ctx.stroke();
```

What makes this API usage pattern more than just a code snippet is that it gives rationale for each part based on the API's execution facts and the concepts the API is based on (e.g., explaining why a stroke or fill is also required) and it conveys how the code might be changed to produce a related behavior (e.g., it explicitly says that a fill could have been used instead to fill in the middle of the rectangle, but also implicitly suggests through each function call that parameters might be changed to achieve different behavior).

Consider other examples from our ThreeJS scenario (Figure 3.2):

- To add a shape to a scene (stage B), developers must first create a Geometry and a Material, then create a Mesh object from those, and then add the Mesh object to the scene. An API usage pattern for this process would show: (i) the order in which to create objects for a Mesh, (ii) that ThreeJS can be used for adding shapes to scenes, and (iii) where to change the code to add different shapes to a scene.

- To make an object appear reflective (stage G), developers must create a `CubeCamera`, add it to the scene, update it (to render), then set the material of the reflective object to use the camera as an environment map with `CubeReflectionMapping`. For example, Figure 3.5 shows an API usage pattern which shows (i) the sequence of function calls for completing this task, (ii) that ThreeJS objects can be reflective, and (iii) where to change the settings of the `CubeCamera`.

- To animate an object, a developer must store it in a variable with a high scope, then in the callback argument of `requestAnimationFrame` (a function provided by JavaScript), they can get the current time and set the position or rotation of the object. An API usage pattern for this task would indicate: (i) that animating ThreeJS objects is possible, (ii) where to put code for animation and how the code should access the object to animate, and (iii) where and how to change the code to change how the object animates.

Prior work shows that code examples are essential to API learning, whether pseudocode that foregrounds usage rules and patterns or full executable examples [73, 106, 59, 126, 30, 110, 115, 88, 140, 118, 146]). However, evidence also shows that it is essential for API usage patterns to explain usage rules and their rationale [31] and why the API was designed in the way it was [73, 82, 110, 57]. Prior work also shows that API usage patterns are key to revealing the range of behaviors that an API makes possible. Examples define a single point in a design space, while an API usage pattern shows a range of possibilities. A developer must have a rich mapping between the behaviors that an API enables and the code that implements them in order to overcome selection barriers [110, 59].

3.3.2 Necessary, sufficient, and exhaustive?

In order to address whether these three components of API knowledge are necessary, sufficient, and exhaustive, we must take into account that there may be a near infinite number concepts, facts, and patterns about an API that may or may not be relevant in any given situation.

Necessary

We certainly don't claim that the near infinite number of concepts, facts, and patterns of an API are all necessary for working with it. In fact a developer may be able to get by while missing knowledge about relevant concepts, facts, and patterns, but any of these that are missing will reduce the robustness of their knowledge about their interactions with code. In fact, if developers are missing knowledge of a particular concept, fact, or pattern, they may be able to infer it from the other components of knowledge:

- *Inferring domain concepts*: A developer might look at a fact or pattern that uses a term they hadn't known before and infer the term's conceptual meaning.

- *Inferring execution facts*: A developer might recognize terminology used in an API element name and guess at how its output relates to the known concept. Alternatively, a developer could look at how and why an API element is used in an API usage pattern and infer some of the execution rules around inputs, outputs, and side-effects of the API element.

- *Inferring usage patterns*: A developer could take a domain concept they know around how to achieve some goal, and then use their knowledge of execution facts of the API to piece together and mentally check potential patterns of code until they successfully invent an API usage pattern they had not known before.

Still, we believe that some set concepts, facts, and patterns are the necessary for doing any particular task with an API, even if it just a code snipped (an unexplained, not generalized API usage pattern) that is copied and pasted into code.

Sufficient and Exhaustive

We do claim that from the limited perspective of an isolated developer understanding code that uses an API, considering options for using an API, and writing or modifying code that uses an API to achieve a specific behavior, all needed knowledge beyond knowledge of the underlying programming language (which has its own concepts, facts, and patterns) comes down to the API's concepts, facts, and patterns. Thus, in this context concepts, facts and patterns are sufficient and exhaustive, with other potentially relevant knowledge simply being aides in learning and interacting with concepts, facts, and patterns (e.g., location of documentation, IDE auto-complete functions, code generators).

Still, creating code in isolation is not the only perspective worth considering when working with APIs, and in many other situations concepts, facts, and patterns are neither sufficient nor exhaustive. If your goal is not just create code that would achieve a specific goal, but also to run that code, you may need to know how to install the API and execute code, requiring knowledge of things like operating systems, platforms, and configuration issues. Code is generally written to be understood by other developers, adding readability as an important part of the rationale of API usage patterns, and requiring knowledge about how to work with and communicate with other developers. APIs also exist with in their own active communities. Taking an active role in these larger communities may involve needing knowledge about community standards, where to ask questions, how the API changes over time, how to report bugs or upgrade API versions.

3.3.3 Evaluating our theory

Earlier we made three claims for what a successful theory of API knowledge would do (3.2.1), so here we evaluate our theory by those three criteria.

Unify prior findings on API knowledge into a coherent model

Our theory of API knowledge is not based on a single source of evidence, but on a diverse collection of prior work, not solely conducted by the authors. Our theory incorporates many previous findings: Domain concepts help developers seek and recognize relevant information [31, 58, 59, 146], understand the purpose of code [57, 106], and decide what purpose they want new code to fulfill [59, 139]. API usage patterns demonstrate what is possible with an API [73], help developers understand how API code works together and how to coordinate API use [57, 106, 59], and encode common usage patterns and best practices [110, 146, 122]. Execution facts elucidate the details of the API structure [73], how to use API features [30, 59, 81], how to understand dependencies and relations between API features [31, 30, 81], and details of how code relates to output [59, 57]. We did not incorporate previous findings that cover API-related knowledge we explicitly exclude from our theory: initial setup of an API that involves the environment [73], how to find information about the internal and external behavior of code [59], and programmers' misapplication of knowledge [60].

Capture the internal logic of how parts of API knowledge relate to each other and the role API knowledge plays in helping a programmer work with code that involves APIs

Our theory attempts to explain why concepts, facts, and patterns are central components in robust API knowledge and how those components relate to each other. We argue that in the (admittedly limited) context of an individual reading and writing (but not running) API code, concepts, facts, and patterns are the core components of knowledge that the developer needs. We explain the role of each component of knowledge in understanding and designing code (Table 3.1), and how each component relates and connects to the others (Fig. 3.1. Together these capture how it is that a developer knows (or fails to know) what to search for, what it means, how to adapt it, and how to fix it when it breaks.

Produces falsifiable claims that can be further tested

Our theory of robust API knowledge makes several falsifiable predictions. For example, we claim that knowledge of concepts, facts, and patterns together form robust API knowledge, and thus have a cumulative benefit. That is a testable hypothesis that, if false, would suggest that the interactions we claimed between these three components of knowledge may not actually occur. Another falsifiable prediction is that knowing more facts about an API's behavior should streamline a developers' debugging. If that were not true, it would suggest that having approximate models of API behavior at runtime a priori is not necessarily an important factor in debugging productivity.

Now that we can make testable hypotheses, the next step is to test them. Theories that produce falsifiable claims require a body of evidence in order test, refine, and potentially refute them; many are abandoned, but ultimately lead to better theories. No publication proposing a theory, as this one does, can sufficiently provide this body of evidence to support its claims. It can only begin to.

In the rest of this paper we will present a two studies as a start, testing seven hypotheses generated from our theory. In the first study, we investigate the impact of access to concepts, facts, and patterns on productivity and comprehension. Our second study will look at how StackOverflow content fits our categories of concepts, facts and patterns. These two studies are just a start to validating our theory through empirical studies. More research will be necessary to further test, refine, or refute the theory's claims.

3.4 Study 1: The impact of robust API knowledge

We designed our first validation of our theory to test the effects of providing the three components of API knowledge on programmers' attempts to understand code and adapt code. We specifically derived five hypotheses from the theory for testing:

- **H.1.a** On a task requiring API learning and use, progress will increase when participants have access to each of the three components of API knowledge.

- **H.1.b** On a task requiring API learning and use, progress will increase when participants have access to a larger number of components of knowledge.

- **H.2.a** Learners' self-reported comprehension of the API will increase when they have access to each of the three components of API knowledge.

- **H.2.b** Learners' self-reported comprehension of the API will increase when participants have access to a larger number of components of knowledge.

- **H.3** Participants will report value in concepts, facts, and patterns for tasks involving an API.

To test these five hypotheses, we designed a controlled laboratory experiment that presented a series of web development task requiring the learning and use of several different APIs. We recruited university students and varied participants' access to each of the three components of knowledge in our theory. This design allowed us to begin to understand the causal impact of different components of knowledge on both task progress and self-reported comprehension. In the rest of this section, we describe our method and results, before turning to our second validation study.

3.4.1 Method

Recruitment

Our inclusion criteria for the study were adults with prior knowledge of JavaScript and the HTML and CSS web standards, but no prior knowledge of the APIs that they would learn in the study. We recruited by emailing students at a large public university who had recently taken a web programming class and by emailing a list of CS PhD students likely to have prior knowledge of web development. Prior to participating, we asked participants to self-report their prior knowledge of the web standards and APIs in the study. Of the 54 participants we recruited, 20 identified as female, and 34 as male. Ages ranged from 19 to 37 years old ($\mu = 21.8$). Fourty-six were undergraduate participants, with 1 freshman, 19 sophomores, 21 juniors, and 5 seniors. Eight participants were doctoral students ranging from first year to fourth year.

Tasks

In designing tasks, we sought diverse, ecologically valid uses of APIs that reflected a developer trying to build something with an unfamiliar API. We chose four different JavaScript APIs, attempting to cover different domains of use, as well as APIs with concepts we believed our participants would likely not know. We chose the four APIs and created the following task sets for each one (see Figures B.1 and B.2 for more details):

- **d3.js** (d3js.org) is a data visualization API that requires knowledge of scalable vector graphics (SVG) and concepts related to data visualization such as marks, color theory, and data mappings. The d3.js task set asked participants to improve a visualization of known exoplanets.

- **Natural** (github.com/NaturalNode/natural) is a small natural language processing and information retrieval API. It requires knowledge of linguistic concepts such as syntax, stemming, and parsing. The Natural task set asked participants to add code to a stubbed-in interface, so the program would process the text of Jane Austin novels and users could see words associated with characters and search for characters by word.

- **OpenLayers** (openlayers.org) is an interactive mapping API. It requires knowledge of concepts in geographic information systems such as projection, markers, and layers. The OpenLayers task set asked participants to improve an interactive map by making it be rounded, with latitude and longitude lines, country outlines, and allow users to draw on the map.

- **ThreeJS** (threejs.org), the 3D graphics library we described in the section 3.3, requires detailed knowledge of geometry, physics, and lighting. The ThreeJS task set asked participants to create a loading screen by adding a spinning, shiny ring to an ocean scene.

For each task set, we gave participants several materials. We created starter code and an development environment, which eliminated the need to learn how to install and configure the API. We provided a description of each task in the task set that they were to achieve. Finally, we provided a code example from which they could learn the API, mimicking the discovery of an example in documentation, a blog post, or a StackOverflow answer.

The starter code for each task set ranged from a 30 line JavaScript program to make a simple map for OpenLayers to a 204 line JavaScript program to make an interface to show the results of using the Natural library. We based most of the starter code on examples that were similar to or exactly matched the examples provided in each API's documentation, but with comments stripped out, so we could isolate the effects of the information we provided on concepts, facts, and patterns.[3]

[3] The main exception to this was our Natural starter code, which mostly consisted of code to load the

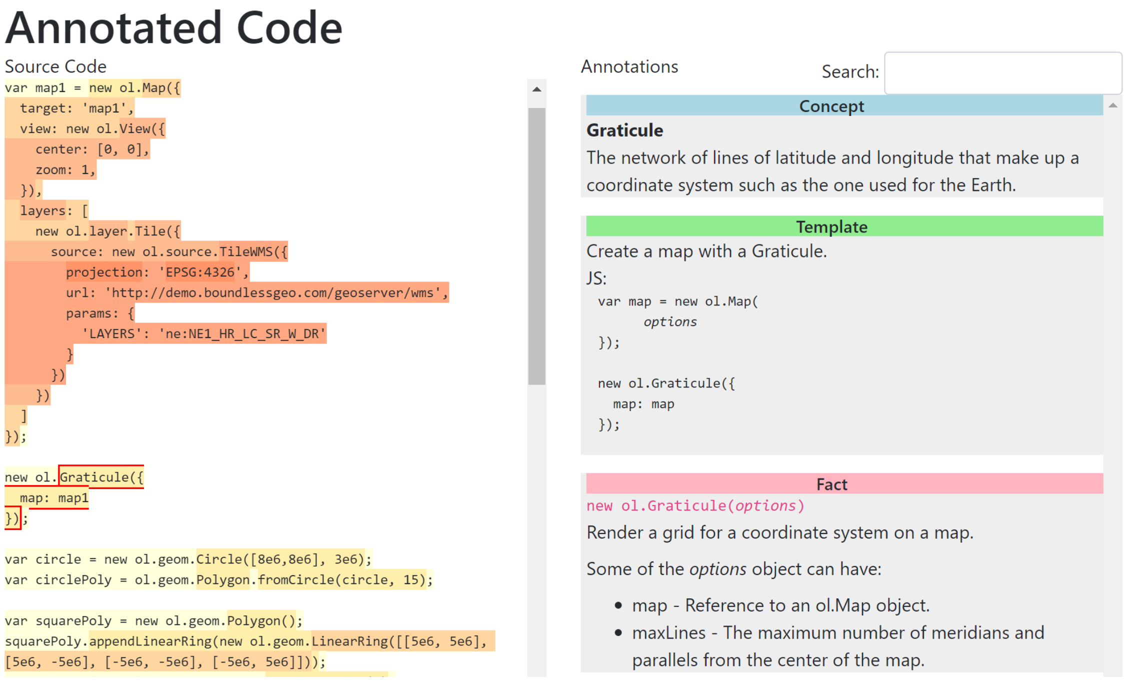
The example code we provided was designed to be fairly close to what the participants needed to complete the tasks, including at least one call to every API function needed to solve the tasks, but were different enough that participants could not directly copy and paste code to solve the tasks. They had to change at least some parameters of function calls to achieve correct behavior. This principle of "near match" mirrored what learners are likely to find online on sites like StackOverflow, but also made the tasks feasible to complete in the short 15 minute task time limits. We also designed example code to minimize the dependence on knowledge of JavaScript, primarily relying on conditional logic, function calls, and simple object literal declarations. This ensured that that participants' progress would depend primarily on understanding of the API, and not on potentially brittle knowledge of JavaScript's language semantics.

We piloted each task with multiple participants with sufficient prior knowledge, clarifying unclear instructions, adding missing annotations, and adapting task difficulty.

Manipulating access to API knowledge

The core manipulation of our experiment was varying access to our theory's three components of API knowledge. To do this, we formatted the examples that participants received with *annotations* on each line of code with the three components of API knowledge. For example, when the code referenced a *concept* that we believed many participants would not know (e.g., pack graph, torus), we wrote a definition, generally based on Wikipedia or Wiktionary (Figure 3.3 shows an example of one of these definitions). For *API usage patterns*, we looked for multiple lines of code that we judged to work together to achieve some purpose, and created a commented code template that represented the usage pattern (Figure 3.5

book files and run the user interface (see Fig. B.1). We commented the Natural code extensively and indicating where modifications would need to be made. This was intended to allow users to focus more on the Natural API code they would have to add to the interface and not the interface itself. Two other places comments were added were to the ThreeJS task marking "Additional code that you don't need to worry about: creating water, animating water, creating sky, responding to resize," and to OpenLayers indicating the code that created the Sphere Mollweide projection, which used another API (Proj).



```
var map1 = new ol.Map({
  target: 'map1',
  view: new ol.View({
    center: [0, 0],
    zoom: 1,
  }),
  layers: [
    new ol.layer.Tile({
      source: new ol.source.TileWMS({
        projection: 'EPSG:4326',
        url: 'http://demo.boundlessgeo.com/geoserver/wms',
        params: {
          'LAYERS': 'ne:NE1_HR_LC_SR_W_DR'
        }
      })
    })
  ]
});

new ol.Graticule({
  map: map1
});

var circle = new ol.geom.Circle([8e6,8e6], 3e6);
var circlePoly = ol.geom.Polygon.fromCircle(circle, 15);

var squarePoly = new ol.geom.Polygon();
squarePoly.appendLinearRing(new ol.geom.LinearRing([[5e6, 5e6],
[5e6, -5e6], [-5e6, -5e6], [-5e6, 5e6]]));
```

Concept

Graticule

The network of lines of latitude and longitude that make up a coordinate system such as the one used for the Earth.

Template

Create a map with a Graticule.

JS:

```
var map = new ol.Map(
      options
});

new ol.Graticule({
  map: map
});
```

Fact

`new ol.Graticule(options)`

Render a grid for a coordinate system on a map.

Some of the *options* object can have:

- map - Reference to an ol.Map object.
- maxLines - The maximum number of meridians and parallels from the center of the map.

Figure 3.6: Annotated example code from Study 1 OpenLayers task set. Code with annotations are highlighted (darker highlight for more annotations). The example code calling `Graticule` is selected and relevant annotations are displayed on the right.

shows an example template).[4] For each line of example code calling a function, we created function descriptions to teach execution facts about the function's behavior (Figure 3.4 shows an example description of a function's execution facts). We generally pulled *facts* directly from the official documentation, though we removed unneeded parameters from some functions to reduce complexity, especially for functions that had dozens of options like ThreeJS's `MeshPhongMaterial`. As we did our initial piloting, we made some modifications and additions to the annotations to improve them. For example, after seeing pilot users express difficulty comprehending the initial definition of n-grams, we added example n-grams to the definition.

[4] While the example code we gave participants are partial API usage patterns, they lack specific mentions of where parameterization can be made and any rationale behind the design of the example.

Figure 3.6 shows the interface we created to present the code example and its annotations. The tool highlighted code that had annotations and when participants hovered over the highlights or clicked on them, the right panel showed the relevant annotations. For example, in Figure 3.6, the `Graticule` code is selected and annotations are shown for the concept of a Graticule, the template for creating a map with a Graticule, and the facts about the `Graticule` constructor. For times where we gave participants no annotations, none of the code was highlighted and annotation area said "No annotations provided for this task". Our goal in allowing participants to select code and see the annotations relevant to it was to reduce the time and effort spent on information retrieval, that way we could see better the effect of the information itself. The tool allowed us to hide or show annotations by knowledge components so that we could test which type of annotations had which benefits to participants in different experimental conditions.

We tested the effect of the different knowledge components by changing which annotations were visible for which API task sets to participants.[5] Each participant had one API task set with no annotations, one with one annotation type, one with two annotation types, and one with all three annotation types. We counterbalanced which annotations they had and for which API task sets to randomize any learning affects that occurred between tasks Table 3.2 shows our various counterbalanced conditions, following a Latin square pattern. We assigned each of our 54 participants a condition number. We continued collecting some additional data after the initial 48 (twice for each condition) to increase the amount of data we had to work with. While collecting that additional data, we made sure to repeat conditions where there had been a technical difficulty in some of the task sets. In doing this we made sure to have at least two complete sets of data for each task set, though we still used the partial data from the people who had technical difficulties. For each participant we also assigned them a random order of API task sets and a random vertical display order of the annotation

[5] Some participants saw a mislabeled fact which we fixed for future participants. When we tried including it in our statistical models it was never a statistically significant factor, so we exclude it from all models reported in this paper.

	Annotations Types Available for API Task Set			
Cond.	**d3.js**	**Natural**	**OpenLayers**	**ThreeJs**
1	None	C	C-T-F	C-T
2	C	C-T	None	C-T-F
3	C-T	C-T-F	C	None
4	C-T-F	None	C-T	C
5	None	C	C-T-F	C-F
6	C	C-F	None	C-T-F
7	C-F	C-T-F	C	None
8	C-T-F	None	-CF	C
9	None	T	C-T-F	C-T
10	T	C-T	None	C-T-F
11	C-T	C-T-F	T	None
12	C-T-F	None	C-T	T
13	None	T	C-T-F	T-F
14	T	T-F	None	C-T-F
15	T-F	C-T-F	T	None
16	C-T-F	None	T-F	T
17	None	F	C-T-F	C-F
18	F	C-F	None	C-T-F
19	C-F	C-T-F	F	None
20	C-T-F	None	C-F	F
21	None	F	C-T-F	T-F
22	F	T-F	None	C-T-F
23	T-F	C-T-F	F	None
24	C-T-F	None	T-F	F

Table 3.2: The 24 counterbalanced conditions of API and annotations available for Study 1.

types (for when there were more than one type).

Participants were told not use any other resources to complete the task sets. Researchers scanned the room every few minutes to ensure people were on task and not using other resources.

Procedure

At the start of the two-hour session, we gave each participant a $30 Amazon gift card for their participation and an option to leave their email address for a copy of the full solutions after the study was complete.

When participants arrived, we gave each a consent form and a survey including questions on how many programming languages and programming libraries they had learned, as well as a rating of their confidence in debugging JavaScript in Chrome. We then gave them instructions that described how the setup of the coding environment (including the Cloud9 IDE) and the three components of core API knowledge which they would see in annotations. We then gave them a sample task to work on using a fifth API (slickGrid) that included annotated example code, so they could get used to navigating our setup. We told them that when it was time to work on the different API task sets, to complete as many tasks as they could (in order).

For each API task set, we gave participants 15 minutes to complete as many tasks as they could, making a backup copy of their JavaScript file every time they completed a task. After the 15 minutes, we told them to stop working and gave them four minutes to rate their understanding of the code. We then gave them another four minutes to hand-write answers to two open ended questions described in section. Details of our measurements, analysis and results are next in section.

3.4.2 Analysis and Results

Below, after discussing measuring prior knowledge, we discuss the influence of annotations on task set progress (section 3.4.2) and on perceived understanding (section 3.4.2), then we discuss the attitudes that participants reported toward the annotations (section 3.4.2).

Measuring prior knowledge

To control for prior knowledge in our models and to assess inclusion criteria, and guided by evidence that people can somewhat reliably estimate their programming experience [33], we used three self-reported measures of programming experience: First, we asked participants to approximate the number of programming languages they had learned and select one of the following: (1, 2-4, 5-9, 10+).[6] The purpose of this question was to gauge general experience working with different types of code. We expected more diverse programming experience to lead to faster reading, understanding and writing of code. Second, we asked participants to approximate the number of APIs they had learned as one of the following: (1, 2-4, 5-9, 10+).[7] We expected more experience learning APIs to help participants learn new APIs more quickly because they may had encountered concepts, patterns, or facts similar to ones in the study. Third, we asked participants to rate their *confidence in debugging JavaScript in the Chrome web browser* on a seven point Likert-scale, as participants would be able to use the Chrome debugger.

The Effects of Annotations on Task Set Progress (H.1.a and H.1.b)

Measuring task progress As participants worked on their task sets, Cloud9 stored a history of their edits and saves. We had participants mark a task when they thought they had completed it, but we wanted to judge their work for ourselves. Since the tasks in the API task sets were challenging and often involved multiple changes to the code, we identified the necessary subtasks to solve each task (Fig. B.1 and B.2). We then rated how many of these subtasks were complete in participants' final code as well as two saves prior (in case they had just done something to break their code right before we called time, such as deleting a line of code in preparation for rewriting it) by reading and executing the code. We then recorded the most subtasks completed in any of those three versions of their code.

One researcher (Kyle) rated the progress on all tasks of all users, and then another researcher (Sarah Chasins) independently rated the progress on 10%. Our inter-rater reliability check found that 64% matched exactly. The low number was mostly due to the difficulty of rating progress in d3.js (only 38% matched), where few participants found the example code related to the first task, and came up with different solutions. Specifically, there were a number of cases in d3.js where participants solved a task in a way we had not intended (by

[6] We later realized we should have clarified whether HTML and CSS counted as programming languages.

[7] We later realized we should have included 0 in number of APIs.

performing task 1 by just modifying the html or using plain JavaScript instead of the d3.js library to add the title), or not finishing one task before moving on to the next. We excluded these from our d3.js task progress analysis and the researchers redid the progress rating jointly for the remaining d3.js participants. For the other three libraries, 79% progress ratings matched exactly and 93% were within one subtask. We additionally had one researcher rate completion of tasks 1.4, 1.5, and 1.7 in ThreeJS for subtask analysis and when we had another researcher (Sarah) do the same for 10% of participants, there was 100% agreement.

Models of Task Progress In order to measure the effects of providing annotations of the different knowledge components on task set progress we created models for each API. The unit of analysis was a participant / API pair. We excluded 3 participant / API pairs where there was a technical difficulty, and we excluded 16 participant / API pairs for the participants who solved part of the d3.js task set without using d3.js. That left us a total of 187 participant / API pairs. After those exclusions, we had only one participant with no task set progress with any API. The other 52 participants all made progress on at least two APIs. Of the 187 participant / API pairs, 27 made no progress and no participant finished the whole task set for any API. There were therefore no ceiling effects and only moderate floor effects on our progress measure.

To build our model, we used task set progress as the dependent variable, and presence of concepts, facts, and patterns, and how many previous programming languages (PLs) learned, how many previous programming libraries (APIs) learned, and their confidence debugging JavaScript in Chrome as our independent variables. Since our dependent variable (task set progress) was ordinal we used proportional odds logistic regression models. We then calculate the p value from the t values and the degrees of freedom.[8]

For overall progress on task sets, certain components of knowledge helped on certain task

[8] We intended to run analyses of variance (anova) on these models to determine statistical significance (as we will later), but we ran into problems with initial values, which we could fix when directly calling the `polr` function in the `MASS` package, but we could not find a way to do this when using the `Anova` function from the `car` package.

	d3.js			Natural			OpenLayers			ThreeJS		
	β	t	p	β	t	p	β	t	p	β	t	p
Has concept	-2.6	-2.7	.017*	0.30	0.38	.71	0.95	1.6	.12	1.2	2.0	.056
Has pattern	2.1	2.5	.023*	2.3	2.9	.007**	-0.68	-1.1	.26	0.046	0.073	.94
Has fact	-0.63	-0.76	.46	-2.3	-2.6	.012*	-1.5	-2.3	.027*	0.79	1.3	.22
# Learned PLs (linear)	0.38	1.3	.78	0.71	0.57	.57	-1.8	-1.7	.11	18	0.15	.88
# Learned PLs (quad.)	0.90	1.2	.26	-0.47	-0.63	.53	-0.69	0.66	.31	9.9	0.14	.89
# Learned APIs (linear)	2.4	2.3	.034*	0.23	0.31	.76	2.2	2.6	.014*	0.75	0.96	.34
# Learned APIs (quad.)	0.40	0.49	.64	-0.64	-1.0	.31	0.98	1.5	.15	-0.40	-0.61	.54
# Learned APIs (cubic)	1.45	1.97	.069	1.1	1.85	.073	0.070	0.13	.90	-0.086	-0.15	.88
debug confidence (linear)	-0.14	-.012	.90	0.15	0.15	.89	-0.53	-0.58	.57	-0.27	-0.25	.80
debug confidence (quad.)	-0.25	-0.22	.82	-0.38	-0.42	.68	0.41	0.52	.61	0.50	0.49	.63
debug confidence (cubic)	-1.1	-0.96	.36	-2.5	-2.2	.032*	0.052	0.054	.96	-0.56	-0.56	.58
debug confidence (^{4})	2.8	2.44	.028*	0.78	0.8	.42	-1.1	-1.22	.23	-0.17	-0.18	.86
debug confidence (^{5})	0.28	0.35	.73	-0.54	-0.74	.46	-0.23	-0.32	.75	-0.021	-0.030	.97

Table 3.3: Ordinal regression model for task set progress in each API, testing the role of the types of annotations available to participants (H.1.a).

	d3.js			Natural			OpenLayers			ThreeJS		
	β	t	p	β	t	p	β	t	p	β	t	p
# Annotation Types: 1	0.076	0.068	.95	0.081	0.099	.92	-1.8	-2.4	.025*	0.49	0.61	.55
# Annotation Types: 2	0.076	0.078	.94	-1.1	-1.3	.19	-0.49	-0.59	.56	0.95	1.2	.25
# Annotation Types: 3	-0.57	-0.55	.59	0.69	0.85	.40	-1.6	-1.9	.073	2.2	2.5	.017*
# Learned PLs (linear)	-0.80	-0.64	.53	0.60	0.46	.65	-1.1	-0.89	.38	17	0.20	.84
# Learned PLs (quad.)	1.0	1.3	.23	-0.66	-0.83	.41	-0.61	-0.79	.44	9.2	0.19	.85
# Learned APIs (linear)	2.3	2.2	.047*	0.67	0.87	.34	2.3	2.7	.012*	0.69	0.89	.39
# Learned APIs (quad.)	0.56	0.65	.53	-0.60	-0.99	.33	0.034	0.053	.96	-0.64	-1.0	.32
# Learned APIs (cubic)	1.5	2.0	.071*	0.81	1.38	.18	-0.0074	-0.013	.99	0.13	0.24	.82
debug confidence (^{1})	-0.31	-0.31	.77	-0.33	-0.34	.74	0.049	0.055	.96	-0.015	-0.014	.99
debug confidence (^{2})	0.38	0.38	.76	0.38	0.44	.66	0.80	1.1	.30	0.43	0.42	.68
debug confidence (^{3})	-1.92	-1.8	.088	-2.2	-2.0	.050*	-0.79	-0.78	.44	-0.77	-0.77	.45
debug confidence (^{4})	1.6	1.5	.15	1.4	1.4	.16	-0.49	-0.52	.61	-0.16	-0.18	.86
debug confidence (^{5})	0.43	0.52	.61	-0.83	-1.1	.26	-0.28	-0.39	.70	-0.15	-0.21	.83

Table 3.4: Ordinal regression model for task set progress in each API, testing the role of the number of annotations available to participants (H.1.b).

sets, some components of knowledge hurt, and others appeared to have no effect (Table 3.3). We found significant differences in the following cases: concepts decreased progress in the d3.js task set ($\beta = -2.6$, $p < .017$); patterns improved progress in the d3.js task set ($\beta = 2.1$, $p < .023$); patterns improved progress in the Natural task set ($\beta = 2.3$, $p < .007$); facts hurt progress in the Natural task set ($\beta = -2.3$, $p < .012$); and facts hurt progress in the OpenLayers task set ($\beta = -1.5$, $p < .027$). We also found that number of previously learned APIs correlated had a significant positive linear correlation with task set progress in d3.js ($\beta = 2.4$, $p < .034$) and OpenLayers ($\beta = 2.2$, $p < .014$), and debugging confidence had a significant positive quartic correlation with task progress in d3.js ($\beta = 2.8$, $p < .028$) and a negative cubic correlation with task set progress in Natural ($\beta = -2.5$, $p < .032$).

In order to test the role of the number of annotation types a participant had in their task set progress, we created proportional odds logistic regression models for each API, as before. Instead of having independent variables for the presence of each knowledge components, we had a categorical variable for the number of annotation types.

We only found two instances where the number of annotations significantly influenced task set progress (Table 3.4): having all three significantly increased task progress with ThreeJS ($\beta = 2.5$, $p < .016$) and having one (as opposed to 0) significantly decreased progress in OpenLayers ($\beta = -1.8$, $p < .025$). The correlations with number of PLs learned, APIs learned, and debugging confidence correlated in similar ways to before.

Whereas the analysis above considered overall task set progress, some of our tasks had clear bottlenecks. We therefore chose specific subtask bottlenecks (refer to Figs. B.1, B.2) to investigate further. We based our choice of bottlenecks on visual examination of where it appeared that one of the annotation types had made a difference (biasing our results toward significant differences, but highlighting the impact of robust API knowledge) and also on whether the amount of data we had for those steps allowed for analysis. This resulted in five bottlenecks from three of the APIs (potential bottlenecks in d3 weren't clear enough or had insufficient data):

- **Natural N-Gram Subtask** *(completed task 1.1)*: This subtask was to start modifying how nearby words should be to a character's name to be considered. This involved changing a call to the *trigram()* function to a call to the *ngram()* function.

- **Natural TF-IDF Subtask** *(completed task 2.1, for those who finished task 1)*: This subtask was to start grouping text together and find words that were relatively more common in different sets. This involved using the *tfidf* functionality.

- **OpenLayers Graticule Subtask** *(completed task 2.1, for those who finished task 1)*: This task was to start adding latitude and longitude lines; this required the `Graticule` constructor.

- **ThreeJS Torus Subtask** *(completed task 1.5, for those who finished task 1.1-1.4*: This subtask consisted of correctly choosing the *TorusGeometry* constructor to create a shape like the one in the example.

- **ThreeJS Torus Params Subtask** *(1.7, for those who finished task 1.5)*: This subtask consisted of making the Torus look smooth by putting a large number in the fourth parameter of *TorusGeometry* constructor to increase the tubularSegments.

For each of these bottlenecks we removed participants who had not made it to the step before the bottleneck, and then we built a similar proportional odds logistic regression models for each API, as before, though we now use analysis of variance (anova) to compute statistical significance. The unit of analysis was a participant / bottleneck pair and our dependent variable was a Boolean measure of whether the participant passed the subtask bottleneck. We used the same independent variables as we did for overall task set progress.

Table 3.5 shows the resulting models for each of these bottlenecks. We found that on the Natural N-Gram subtask (1.1), patterns (which showed how various blocks of the example code worked, though there were no patterns on N-Gram use in the example code) helped the participants make task set progress ($\chi^2(1, N = 54) = 11.7$, $p < .0006$), while facts (which

	Natural N-Gram (task 1.1)					Natural TF-IDF (task 2.1)					OpenLayers Graticule (task 2.1)				
	model		anova			model		anova			model		anova		
	β	z	df	χ^2	p	β	z	df	χ^2	p	β		df	χ^2	p z
Concept	0.05	0.05	1	0.004	.96	40	0.002	1	3.5	.063	4.1	2.2	1	8.7	.003**
Pattern	3.5	2.7	1	11.7	.0006***	-3.8	0.0	1	0.0	1.0	-1.3	-1.1	1	1.2	.27
Fact	-3.9	-2.9	1	14	.0002***	45	0.001	1	2.8	.096	-0.93	-0.75	1	0.59	.44
# PLs^1	-2.3	-0.92	2	4.0	.13	134	0.002	2	9.9	.007**	-13	-0.006	2	2.8	.24
# PLs^2	-2.3	-1.5				-4.9	0.0				-8.2	-0.006			
# APIs^1	-0.01	-0.07	3	3.0	.39	-5.1	0.0	3	0.68	.88	12	0.005	3	4.7	.20
# APIs^2	0.51	0.55				-0.4	0.0				10	0.006			
# APIs^3	1.3	1.5				12	0.0				3	0.004			
Debug^1	0.43	0.34	5	6.1	.30	-33	-0.001	4	10	.040*	-12	-0.003	5	3.8	.58
Debug^2	-1.3	-1.2				29	0.001				10	0.003			
Debug^3	-2.1	-1.3				0.61	0.0				-5.2	-0.002			
Debug^4	0.09	0.067				-52	-0.001				3.9	0.003			
Debug^5	0.51	0.53				N/A	N/A				-1.6	-0.003			

	ThreeJS Torus (task 1.5)					ThreeJS Torus Params (task 1.7)				
	model		anova			model		anova		
	β	z	df	χ^2	p	β	z	df	χ^2	p
Has concept	2.6	2.0	1	5.4	.02*	-1.0	-0.50	1	0.26	.61
Has pattern	0.55	0.43	1	0.18	.67	-59	-0.004	1	8.5	.004**
Has fact	0.85	0.75	1	0.60	0.44	82	0.005	1	30	$< .0001$***
# Learned PLs (linear)	14	0.007	2	3.5	.17	47	0.001	2	9.6	0.008**
# Learned PLs (quadratic)	8.0	0.007				27	0.001			
# Learned APIs (linear)	2.1	1.6	3	4.6	.20	-45	0.0	3	3.3	.34
# Learned APIs (quadratic)	-1.9	-1.5				-0,99	0.0			
# Learned APIs (cubic)	1.3	1.2				-0.081	0.0			
Debug confidence (linear)	2.3	1.3	5	3.7	.58	-9.8	0.0	5	7.9	0.16
Debug confidence (quadratic)	-0.6	-0.37				5.1	0.0			
Debug confidence (cubic)	-0.59	-0.32				-38	-0.002			
Debug confidence (^{4})	1.3	0.72				24	0.001			
Debug confidence (^{4})	-1.7	-1.36				-16	-0.002			

Table 3.5: Anova for ordinal regression of task set progress by API for specific task / subtask progress. Note that these were chosen based on appearance of annotations making difference based on visual inspection, so the p values are artificially biased toward significance.

showed parameters for the n-gram methods, though didn't explain the meaning of n-grams) hurt progress there ($\chi^2(1, N = 54) = 14$, $p < .0002$). On the Natural TF-IDF subtask (2.1), we found that concepts (which included a definition of TF-IDF) trended (though not significantly) toward helping participants make progress ($\chi^2(1, N = 30) = 3.5$, $p < .063$). On the OpenLayers Graticule subtask (1.1) we also found that concepts (which included a definition of graticule) also helped participants make progress ($\chi^2(1, N = 45) = 8.7$, $p < .003$). On the ThreeJS Torus subtask (1.5), we found that concepts (which included the definition of Torus shown in Fig. 3.3) helped participants make progress ($\chi^2(1, N = 47) = 5.4$, $p < .02$). Finally, on the ThreeJS Torus Params subtask (1.7), we found that facts (which included the parameters of TorusGeometry shown in Fig. 3.4) helped participants make progress ($\chi^2(1, N = 36) = 30$, $p << .0001$), while patterns hurt progress ($\chi^2(1, N = 36) = 8.5$, $p < .004$). We did not find that the number of APIs previously learned had any significant influence on these specific bottlenecks.

The Effects of Annotations on Perceived Understanding (H.2.a and H.2.b)

Measuring Understanding To measure how well participants learned the API at the end of their task set, we chose to measure their perceived understanding of the example code, since that code was still the same for all participants at the end of the task set and included many calls to the APIs. After the each task set we gave participants a paper copy of the example code, and for each substantive line of code (we marked these for them), to rate their agreement with the statement "I understand what this line does and its purpose in the larger program."[9] on a five-point Likert scale. Not all participants finished rating the whole example code within the time. We would have liked to measure this understanding before the task set as well, but we didn't want to use the time.

[9] In our pilot study, we tried asking separately about how well they thought they understood what the line does, and how well they understood the larger purpose in the program, but participants found it confusing and difficult to answer.

Models of Understanding In order to measure the effects of providing annotations of the different knowledge components on perceived understanding, we created models for each API task set. The unit of analysis was an individual line's understanding ratings. Since some participants did not finish line ratings (or missed some), we excluded data on lines that more than 10% of participants did not rate. We used line knowledge rating as the dependent variable and our independent variables were the presence of concepts, the presence of patterns, the presence of facts, number of programming languages previous learned, how many previous programming libraries (APIs) the participant said they had learned, and debugging confidence. We also set line number as a random effect to account for us asking each participants about multiple lines. We used a cumulative link mixed models since the dependent variable (line knowledge rating) was ordinal and we had a random effect (line number). We then ran analyses of variance on these models to determine statistical significance.

We found a number of cases where each of concepts, facts and patterns (ignoring whether they other ones were present) either increased or decreased perceived understanding in the four different API task sets (Table 3.6). We found that concepts increased perceived understanding in ThreeJS ($\chi^2(1, N = 1927) = 23$, $p << .0001$), and decreased perceived understanding in Natural ($\chi^2(1, N = 1334) = 6.3$, $p < .01$) and OpenLayers ($\chi^2(1, N = 1850) = 13$, $p < .0002$). Patterns increased perceived understanding in d3.js ($\chi^2(1, N = 1391) = 9.0$, $p < .003$), and Natural ($\chi^2(1, N = 1334) = 32$, $p << .0001$), and decreased perceived understanding in OpenLayers ($\chi^2(1, N = 1850) = 15$, $p < .0001$). Facts increased perceived understanding in ThreeJS ($\chi^2(1, N = 1927) = 4.4$, $p < .04$) and decreased perceived understanding in d3.js ($\chi^2(1, N = 1391) = 5.8$, $p < .02$). We also found that number of previously learned programming languages always had an effect and was generally a positive one. Number of previously learned APIs, and debugging confidence were also always significant, but whether it correlated with more or less perceived understanding was inconsistent.

To test the role of the number of annotation types a participant had in their perceived understanding, we created a cumulative link mixed model for each API task set as before. We had a categorical independent variable for the number of annotation types instead of the

	d3.js					**Natural**				
	model		anova			model		anova		
	β	z	df	χ^2	p	β	z	df	χ^2	p
Has concept	-0.042	-0.35	1	0.12	.73	-0.33	-2.5	1	6.2	.01*
Has pattern	0.35	3.0	1	9.0	.003**	0.72	5.7	1	33	<.0001***
Has fact	-0.28	-2.4	1	5.8	.02*	0.068	0.51	1	0.26	.61
# Learned PLs (linear)	0.87	4.2	2	30	<.0001***	1.2	5.3	2	38	<.0001***
# Learned PLs (quad.)	-0.026	-0.22				0.17	1.3			
# Learned APIs (linear)	0.082	0.55	3	13	.005**	-0.15	-0.93	3	45	<.0001***
# Learned APIs (quad.)	-0.40	-3.4				0.24	1.9			
# Learned APIs (cubic)	0.11	1.1				0.63	5.6			
debug confidence (linear)	1.2	6.3	5	62	<.0001***	0.37	1.9	5	101	<.0001***
debug confidence (quad.)	-0.8	-4.9				0.13	0.76			
debug confidence (cubic)	0.078	0.40				-1.4	-6.6			
debug confidence (^{4})	0.46	2.6				1.3	6.9			
debug confidence (^{5})	-0.15	-1.2				-1.3	-8.7			

	OpenLayers					**ThreeJS**				
	model		anova			model		anova		
	β	z	df	χ^2	p	β	z	df	χ^2	p
Has concept	-0.37	-3.6	1	13	.0002***	0.47	4.8	1	23	<.0001***
Has pattern	-0.37	-3.8	1	15	<.0001***	-0.20	-1.9	1	3.8	.05
Has fact	-0.11	-1.0	1	1.1	0.30	0.20	2.1	1	4.4	.04*
# Learned PLs (linear)	0.9	5.2	2	33	<.0001***	0.97	5.3	2	30	<.0001***
# Learned PLs (quad.)	0.11	1.1				0.41	3.8			
# Learned APIs (linear)	0.38	3.0	3	62	<.0001***	-0.15	-1.1	3	14	.003**
# Learned APIs (quad.)	-0.42	-4.0				-0.33	-3.0			
# Learned APIs (cubic)	0.66	7.1				-0.13	-1.3			
debug confidence (linear)	0.081	0.53	5	26	<.0001***	0.45	2.7	5	54	<.0001***
debug confidence (quad.)	-0.45	-3.4				-0.085	-0.55			
debug confidence (cubic)	0.045	0.29				0.18	1.1			
debug confidence (^{4})	-0.21	-1.5				-0.48	-3.1			
debug confidence (^{5})	-0.35	-3.1				-0.41	-3.5			

Table 3.6: Ordinal mixed model regression model and anova for perceived understanding in each API, testing the role of the types of annotations available to participants (H.2.a).

presence of annotation type. Everything else about the model was the same as before.

We found that having more than zero annotations often improved understanding, but understanding wasn't necessarily improved when there were more than one type present (Table 3.7). For all four models, the number of annotations was significant: d3.js ($\chi^2(3, N = 1391) = 61$, $p << .0001$), Natural ($\chi^2(3, N = 1334) = 38$, $p << .0001$), OpenLayers ($\chi^2(3, N = 1850) = 52$, $p << .0001$), and ThreeJS ($\chi^2(3, N = 1927) = 18$, $p < .0004$). In d3.js, having one or two annotation types showed an improvement in understanding, but having three annotation types decreased understanding. In Natural, having one, two, or three annotation types all showed improvement in understanding, but having three showed the least improvement. In OpenLayers, having one, two, or three annotation types showed decreased understanding, and more so with each additional annotation type. In ThreeJS, having one, two, or three annotation types all showed improvement in understanding, and more so with each additional annotation type. The fact that we saw relatively smaller coefficients for having three annotation types with three of the APIs, points to the possibility that at some point information became overwhelming for our participants, as we'll see below in 3.4.2.

Developers' Attitudes Toward Annotations (H3)

To test our hypothesis that participants would perceive value in the three components of knowledge, after each task set we asked participants: "What information or strategies did you find most helpful?" and "What additional information would you have wanted?" These questions also allow us to identify other types of knowledge not in our theory. Participants wrote their answers by hand and we transcribed these for analysis.

One researcher started with codes for the three components of knowledge from our theory, and then did open coding to produce codes for additional knowledge and strategies, and how students valued those. We combined codes for the answers to the two questions to produce a set of codes for each participant/API pair. Once we finished coding, we had another researcher independently code code 10% according to the code book, and we found that 78%

	d3.js					**Natural**				
	model		anova			model		anova		
	β	z	df	χ^2	p	β	z	df	χ^2	p
# Annotation types: 1	0.18	1.2	3	61	<.0001***	0.85	5.3	3	38	<.0001***
# Annotation types: 2	0.90	5.9				0.83	5.3			
# Annotation types: 3	-0.19	-1.2				0.53	3.3			
# Learned PLs (linear)	0.77	3.8	2	29	<.0001***	0.86	3.8	2	27	<.0001***
# Learned PLs (quad.)	-0.026	-0.22				0.0091	0.068			
# Learned APIs (linear)	0.13	0.91	3	26	<.0001***	-0.074	-0.47	3	59	<.0001***
# Learned APIs (quad.)	-0.56	-4.7				0.42	3.3			
# Learned APIs (cubic)	0.25	2.3				0.73	6.4			
debug confidence (linear)	1.2	6.3	5	65	<.0001***	0.15	0.76	5	109	<.0001***
debug confidence (quad.)	-0.83	-5.0				0.22	1.3			
debug confidence (cubic)	0.016	0.08				-1.5	-7.2			
debug confidence (^{4})	0.49	2.8				1.4	7.4			
debug confidence (^{5})	-0.18	-1.4				-1.3	-8.9			

	OpenLayers					**ThreeJS**				
	model		anova			model		anova		
	β	z	df	χ^2	p	β	z	df	χ^2	p
# Annotation types: 1	-0.26	-2.0	3	52	<.0001***	0.28	2.1	3	18	.0004***
# Annotation types: 2	-0.32	-2.4				0.30	2.3			
# Annotation types: 3	-0.94	-7.0				0.55	4.2			
# Learned PLs (linear)	0.85	4.9	2	29	<.0001***	0.97	5.3	2	30	<.0001***
# Learned PLs (quad.)	0.16	1.5				0.40	3.7			
# Learned APIs (linear)	0.40	3.1	3	53	<.0001***	-0.15	-1.1	3	24	<.0001***
# Learned APIs (quad.)	-0.42	-4.0				-0.47	-4.3			
# Learned APIs (cubic)	0.58	6.0				-0.060	-0.60			
debug confidence (linear)	0.13	0.86	5	24	<.0001***	0.54	3.1	5	54	<.0001***
debug confidence (quad.)	-0.43	-3.2				-0.055	-0.35			
debug confidence (cubic)	-0.043	-0.23				0.067	0.40			
debug confidence (^{4})	-0.24	-1.6				-0.41	-2.7			
debug confidence (^{5})	-0.33	-3.0				-0.45	-3.9			

Table 3.7: Ordinal mixed model regression model and anova for perceived understanding in each API, testing the role of the number of annotations available to participants (H.2.b).

of codes matched.

We will report the results of our coding along with specific comments below. We will first report on annotations, including comments on each knowledge components, and then we will report on other sources of information. When reporting below, we select quotes from the comments that were more informative, since most lacked any details (e.g., "*The concept information is helpful*" P67, d3.js; "*[I would like] three.js documentation*" P26, ThreeJS).

Concepts When participants were given concept annotations, they often reported them being useful (27 participant/API pairs out of 106), while none commented on the concept annotations being useless or of little help. Those who gave details as to how concept annotations were useful mentioned two ways they helped. P32 and P42 used concepts to figure out which function to call; P32 referred to ThreeJS Torus subtask, saying, "*Having pictures under the geometry helped me find the correct shape,*" and P42 referred to the Openlayers Graticule subtask, saying, "*'graticule' was an unfamiliar word that I didn't find until control-F ing for 'lattitude' [and finding the definition of Graticule].*" P33, on the other hand, found concepts helpful for remembering in the Natural task set, saying, "*I thought the concepts were really helpful since I sort of forgot what n-grams were.*"

Some participants who had concept annotations wanted more or better definitions (13 participant/API pairs out of 106). For example, P4 wanted additional concept explanations in the OpenLayers task set, some related to the tasks, some not, saying they wanted, "*What some of the codes meant (ESRI, ESPG, etc.). More defined words (ex: extent, worldextent, Projection, Stroke, etc.)*" Others wanted clearer definitions, such P14 with the Natural task set, asking for "*a more thorough explanation of tfidf (how it works, different applications, etc.).*"

When participants were without concept annotations, they often reported wanting concept definitions (34 participant/API pairs out of 108). For example, in ThreeJS, P43 suggested specific concept annotations that we had given to others, saying they wanted "*general information about what shapes I could make and the names of those shapes (possibly pictures* *with corresponding names).*" Similarly P61 in the Natural task set, suggested an annotation we showed other participants, saying, "*It would have been more useful to have an explanation of what TFIDF is used for and what you may want to learn from using it.*."

API usage patterns When participants were given API usage pattern annotations (in the form of templates), they often reported them being useful (23 participant/API pairs out of 108). For example, P67 said the pattern for adding an object to the was scene helpful in ThreeJS, but didn't specify how. Two participants were a little more specific with how patterns helped: P30 in d3.js said, '*I also liked having a template to work off of.*" and P35 in ThreeJS said, '*Template also provides a general grasp on the structure of the code.*"

Only a few participants who had pattern annotations wanted more or better patterns (4 participant/API pairs out of 108). For example, P16, in d3.js, was confused by an italicized term *tagname* in the pattern (we intended this to indicate that an HTML or SVG tag name belonged in that spot), and P54 in OpenLayers wanted a pattern that more clearly laid out how to solve task 4.

A few more participants who had pattern annotations commented on the pattern being useless or of little help (6 participant/API pairs out of 108). Among those who said they did not find patterns helpful, P60 in Natural did not solve the first task (for which there was no helpful pattern) saying, "*I wasn't able to use templates much because I was still struggling to figure out [task 1]*" and P61 had trouble making sense of some details in the patterns in d3.js, though they had all annotations available, saying, "*The template annotations were not very helpful because I did not understand the classes and methods, though the comments in the templates were helpful.*"

When without pattern annotations, some participants reported wanting pattern information (10 participant/API pairs out of 106). Of those 10, 9 used the term "template" (as we had in the instructions and interface) in saying what they wanted. For example P23 in ThreeJS requested a pattern showing how to make an object reflective (a task they did not complete), a pattern we gave to others, and P33 in ThreeJS requested a pattern for how to add an object to a scene (a task they completed), again, a pattern we gave to others.

The code examples themselves can also be considered as partial patterns (not parameterized and without explicit rational), but we will cover that in its own section below.

Facts When participants were given fact annotations, they often reported them being useful (26 participant/API pairs out of 108). For example, in Natural, P33 said, "*I was able to complete the first task once I found the .ngrams(string, # of words).*" Another participant, P38, in ThreeJS (without clarifying for which task), said, "*It was SUPER helpful to have a the explanation of each individual parameter for a given annotations so you could figure out what to change to make it work.*"

Many participants who had fact annotations wanted more or better facts (38 participant/API pairs out of 108). For example, P18 in ThreeJS wanted more than the written description we provided, saying, "*[I would want] the illustration and effects for the parameters in function call. I wasted a lot of time on learning how to make a perfect circle the 'TorusGeometry.'*" P16, on the other hand, wanted a detail we didn't include in the fact annotation, which they needed in OpenLayers task 3.7, saying "*I'm not sure how to set color in hex '#3399cc', so I used rgba [instead].*"

Only three participants who had fact annotations commented on the facts being useless or of little help (out of 108 participant/API pairs): P21 in Natural pointed out an error in one of our fact annotation relevant to task 1, P4 in d3.js said, "*facts were not very useful w/o the templates*", and P7 in d3.js had trouble with getting relevant facts through the interface.

When lacking fact annotations, participants reported wanting them (58 participant/API pairs out of 106). Of those 58, one used the term "fact" in saying what they wanted; the other 57 asked for something we consider a fact. For example, P6 in d3.js wanted fact annotations matching what we gave others: "*[I would want] descriptions of each d3.js function (ex: d3.json, d3.hierarchy).*" P57 talked about their difficulty in completing ThreeJS tasks 1.5-1.7:"*[I want] actual documentation of the Torus constructor - default initialization with 1 argument was all that was present in the example code. I had to guess to find the inner* *radius, and was unable to locate the parameter for number of segments.*'

Annotations in combination Two participants mentioned how the the annotation types worked together: P4 was already quoted ("*facts were not very useful w/o the templates*"). P61 commented on two API task sets: in OpenLayers saying, "*The templates and example code gave enough information where I didn't suffer too much from a 'what can I type?' issue, but they didn't explain meaning at all*," and in d3.js, saying, "*The facts were helpful, but they used terms I did not understand.*"

All the code, instructions, and annotations could be overwhelming as well, especially when participants only had 15 minutes to work with each API. Eight API/participant pairs mentioned there being too much information on the screen (1 had 1 annotation type, 4 had two annotation types and 3 had all 3), saying the information cluttered the screen, was intimidating, or that they wanted shorter descriptions. Two participants wanted the annotations in a different order (something we randomized): P47 in Natural didn't want concepts at the bottom, and the P23 in d3.js didn't like that facts and concepts "were buried under templates." In addition, 12 API/participant pairs mentioned that time was a limiting factor or that they wanted more time.

Example Code (partial API usage patterns) We already reported the comments on pattern annotations, but the code examples we gave participants may be considered partial API usage patterns. This example code we gave them is like information they might find online, which sometimes does not include comments or information about parameterization.

In many task sets, participants mentioned that they made use of the example code or found it useful (115 participant/API pairs of 214). Participants said they used example code in several ways:

- Seventeen API/participant pairs mentioned copying-and-pasting code from the example. For example, P13 in OpenLayers said, '*The first task was essentially copy & paste. Without understanding why it work or what it was doing in depth.*" In contrast, P24 mentioned modifying the code that was copied-and-pasted while working on ThreeJS task 1, *"Copying the example code to my JS program, which I knew would produce some specific shape, and then playing around with parameters for shape and material construction to try and get my desired result."*

- Participants were sometimes able to infer information from the example code. For example, four participants compared code from the two example maps in OpenLayers, which showed different input construction (implying API usage patterns), and the effects of those inputs (implying facts). Additionally, there were 12 mentions of variable or function names being useful. These may have helped participants infer concepts or facts about the API, such as P21 in d3.js, saying, "*Some names in the source code were helpful to make educated guesses about how I might complete a task (i.e. "r" probably having something to do with a radius).*" There were also six mentions of wanting better variable or function names, such as P51 in d3.js saying, "*Info regarding poorly-named vars ('g' in particular) would have been nice as well.*"[10]

- Some participants mentioned wanting more from examples (35 participants/API pairs of 214), such as P32 asking for another example map in OpenLayers, P20 wanting example code closer to the task, or P69 wanting to modify and test the example code in d3.js and ThreeJS.

- Two participants said example code with no annotations and a brittle understanding was limiting. P61 in ThreeJS, said, "*Comparing to the example code was useful for duplicating exactly what it did. I found it very difficult to extend the example code to something new [emphasis original].*" and P19 in ThreeJS said they did not "*know how to do anything outside of the given example. Could only replicate shapes and change values, not make new shapes.*"

[10] "g" is the SVG tag name for a group, which is passed in to d3.js to tie SVG elements together.

Other Sources of Information There were a number of other places participants found information:

Five participant/API pairs mentioned using previous conceptual knowledge, such as P7 in Natural saying, "*I already knew about TF&IDF because we implemented it in [a CS course],*" and P57 in ThreeJS saying, "*[I used] prior knowledge of scene graph-based rendering scheme.*" Two participant/API pairs mentioned how they didn't have prior conceptual knowledge, such as P21 in OpenLayers saying, "*It may have helped if I were more familiar with maps.*" Nine participants also mentioned JavaScript or other web programming knowledge: one said it helped that the starter code for the Natural API was familiar; eight wanted more, all of them when working with d3.js, such as P11, saying, "*[The] API requires really strong pre-requisite knowledge of JavaScript as it was hard to grasp.*"

Another way participants gleaned information was through experimentation. There were 48 mentions of experimentation (out of 214 participants/API pairs), such as P4 in d3.js, saying what they found useful was, "*modifying parts of my code and checking what the result was,*" or P18 in Natural saying the strategy they used was"*print debugging.*"

Finally, one participant (P23) said they figured out the TorusGeometry parameters (facts) without the annotations by taking the exceptional step of examining the minified ThreeJS source code.

Discussion

Table 3.8 summarizes the results of our regression models for annotation types from Study 1. Overall we found support for our hypotheses, with several exceptions:

- H.1.a: For many tasks, access to information about concepts, facts, and patterns was associated with more progress, but for others, it was associated with less.

- H.1.b: Annotations only showed a cumulative positive effect in one API (ThreeJS) where more annotation types meant more progress.

	Beneficial		**Detrimental**	
Annotation Type	**Task Progress**	**Perceived Understanding**	**Task Progress**	**Perceived Understanding**
Concepts	OpenLayers Task 2.1 ThreeJS Task 1.5	ThreeJS	d3.js Overall	Natural OpenLayers
Patterns	d3.js Overall Natural Overall Natural Task 1.1	d3.js Natural	ThreeJS Task 1.7	OpenLayers ThreeJS
Facts	ThreeJS Task 1.7	ThreeJS	Natural Overall Natural Task 1.1 OpenLayers Overall	d3.js

Table 3.8: Summary of regression results for annotation types in Study 1.

- H.2.a: For many tasks, access to information about concepts, facts, and patterns was associated with increased perceived understanding of the API, but for others, it was associated with less perceived understanding.

- H.2.b: There were mixed results on whether there was a cumulative effect: one API (ThreeJS) had improvements with each additional annotation type present, OpenLayers had detriments with each additional annotation type. The other two showed reduced benefits when all three were present.

- H3: Participants' sentiments toward the annotations were largely positive, and when participants lacked a particular component of knowledge, they expressed wanting it. However, they also indicated not understanding some annotations, and when having multiple annotations, feeling overwhelmed trying to understand them during the short 15 minute tasks.

We observed several factors in specific that help interpret these findings:

- *d3.js*: Task 1 depended on finding the code that added text at the bottom of the example code. The participants had to scroll down and many likely missed it (e.g., P61 said, "Now I see there is an example for the first task. I needed to scroll down and did not see it during the exercise."). There were also no concept annotations relevant to the first task (concepts hurt progress), though there was patterns relevant to that task (patterns helped progress). There was also an important fact in the middle of a fact description (how the spacing was calculated for the circles based on the hierarchy "sum" value) which we think was difficult to notice.

- *Natural*: The starter code was long and required some effort to understand, though we tried to alleviate this by pointing participants to the right section of code. The example code was a bunch of calls to the API (taken from the official documentation), and we think the patterns may have helped participants to more quickly focus on code segments.

- *OpenLayers*: The example code consisted of two short examples next to each other which showed all the features needed. This seemed to be sufficient for inferring many concepts, facts, and patterns, and so other than a definition of "Graticule" for Task 2.1, additional information just slowed people down.

- *ThreeJS*: The code was long and depended on concepts that were likely less familiar (torus, specularity, Phong material, etc.). In our example `TorusGeometry` call, we intentionally left out the smoothing parameters that would need to be called (though we included an example of the smoothing parameter on the `SphereGeometry` example). To make it look right, developers needed to both choose the right shape (Torus) and modify the right parameters. This two step process let developers mistakenly go down multiple wrong paths.

Our results provide somewhat noisy support for our hypotheses derived from our theory of robust API knowledge. We found some cases where each type provided statistically significant help, but also cases where it provided significant harm, or had no significant effect. We also learned that many participants valued the three components of knowledge, but pointed out various failings in the information we provided (e.g., not clear, not relevant to their task, not enough information).

This leads us to believe that these three components of knowledge *can* significantly impact task progress, but only the right pieces of that knowledge in the right conditions. They also reveal what these conditions might be: simply having access to concept definitions, patterns, and execution facts is not sufficient; using them fruitfully depends on getting the right piece of knowledge, the quality of instruction that teaches this knowledge, the amount of time that people have to learn it, and the ability to find that knowledge in the medium conveying it.

Future work should better control for these factors, developing higher quality instruction for each knowledge component, providing more effective ways for finding the right piece of instruction, and providing enough time for participants to acquire however much knowledge they need to learn. The results of such a study would either show clearer patterns of positive impact on API use, supporting our theory, or, if they revealed similarly inconsistent impact, suggesting there are other factors that dominate successful learning and use of APIs which our theory doesn't account for.

3.5 Study 2: Content analysis of StackOverflow Q&A

Study 1 tested the impact of *access* to robust API knowledge on task progress and perceived API comprehension, finding some support for the importance of the theory's three types of knowledge. However, it only considered this for specific tasks across four APIs in one programming language, and with a small group of relatively inexperienced developers. To broaden our initial validation of our theory, this second study tests two additional hypotheses in the more ecologically valid learning setting of Stack Overflow (or SO, a site others have studied the content of as well [47, 6]).

Our theory implies several claims about SO. When developers get stuck working with an API, our theory claims that the cause is brittle knowledge of the API's concepts, patterns, or facts. To make progress, they need other pieces of API knowledge (or knowledge of topics like installation or underlying language details). The people answering questions on SO are likely to be developers with more robust API knowledge and are therefore likely to write answers containing what they consider to be the relevant pieces of robust API knowledge.

Since the aim of SO is to help people asking questions get unstuck, answers should contain only relevant API knowledge given the presumed prior knowledge of the questioner and other readers, and not more information. Furthermore, any future person with the same question reading a SO post is likely to upvote questions, upvote answers, or comments, signalling a knowledge gap.

If the theoretical claims above are true, then we predict the following hypotheses to be true:

- **H.4** The majority of the top-voted answers on SO will contain information from one or more of the three knowledge types in our theory. The rest of the information will be knowledge types explicitly excluded from our theory (e.g., installation, location of resources).

- **H.5** The majority of the comments on top-voted answers on SO will contain information from one or more of the three knowledge types in our theory. The rest of the information will be knowledge types explicitly excluded from our theory.

If concepts, patterns, and facts are indeed the three components of robust API knowledge, and if we expect that the best SO answers aim to impart robust knowledge, then we should also expect that these knowledge types will predominate in the best answers and content of comments. If our theory's are not true, we should find many additional categories of critical knowledge, which would require refining our theory.

To test these hypotheses, we focused our attention on top-voted answers because although we cannot be sure they offer the *best* answer to any given question, we can be confident the community considers them *good* answers. We also analyzed replies to the top-voted answer because they often serve as an extension of the answer and also provide additional insights into what the answer lacks. Comments that add facts, concepts, patterns, or other knowledge improve the answer and may elicit a portion of the answer's votes. Comments that request

Language	**API**	**Uses**
Java	Gson	Serialization and deserialization
C	BLAS	Basic Linear Algebra Subprograms; building blocks for vector, matrix operations
C++	Eigen	Template library for linear algebra; matrices, vectors, numerical solvers, and related algorithms
Python	multiprocessing	Process-based "threading" interface
C#	ASP.NET Core	Framework for building cloud-based, Internet-connected applications
Visual Basic .NET	LINQ	Language-Integrated Query; query syntax for retrieving data from different sources, including databases
PHP	PHPExcel	Read from and write to Excel spreadsheet file formats
Ruby	kaminari	Pagination
R	ggplot2	Data visualization
Delphi/Object Pascal	JEDI Code Library	Utility functions; strings, files and I/O, security, math

Table 3.9: We studied the contents of the top SO posts for these 10 libraries in 10 popular languages.

facts, concepts, patterns, or other knowledge serve to reveal a gap between the answer's contents and the needs of the viewers.

One danger with evaluating our knowledge types by assessing SO content is that we have the freedom to make the three knowledge categories vague enough to classify all answer content in them. Our solution was to classify content with categories at a smaller granularity than our theory, then assess whether our fine-grained categories fit within the fact, concept, and pattern knowledge types. For instance, in the data we will describe, there were 12 mentions of code style or readability. Our theory does not account for this, but practical knowledge of an API, especially for reading others' code, may include knowing the preferred idioms of the API community.

3.5.1 Method

To study whether good SO answers focus mostly on elements of robust knowledge, we selected a diverse set of 10 APIs. We started with the list of the most-used programming languages according to the TIOBE index [133]. We excluded SQL because no popular APIs use SQL as a host language, and we excluded JavaScript because Study 1 evaluated JavaScript API knowledge. Following these exclusions, the selected languages seen in Table 3.9. By varying the host language, we attempt to evaluate across a diverse audience of software developers. To choose a diverse range of API abstractions, we selected APIs for a variety of different tasks and usage scenarios, one API per target language (see Table 3.9).

For each API in our target set, we chose the five questions tagged with the API's name with the most votes. In selecting these five, we excluded questions that relate to the host language rather than the target API (7 questions), questions for which the top answer uses a different API (2 questions), questions that focus on comparisons of the target API and other APIs (2 questions), one question about the API's target domain rather than the API, one question about completing a non-API task with the output of the target API, one question about the programming environment rather than the target API, one question about the maintenance of the library rather than the API content, and one question about using version control rather than API content. These exclusions are necessary because question askers can tag questions with an API name even if their question is not specific to the API or even if the answers do not use the API.

For each question in the 50-question set that resulted from our selection process, we collected the text of the top-voted answer and the text of all comments responding to the top-voted answer. The final dataset included 50 answers and 336 comments.

We used open coding – with awareness of our three API knowledge types – to produce fine-grained categories for the information in each text, though in the SO context we coded for "examples" rather than "patterns," since that is how they are generally referred to there. The end result is an analysis of which answers and comments included facts, concepts, and patterns, and which included 25 other types of information that did not fit within our theory of robust API knowledge.

Theory Knowledge Type	Ans	%Ans	Com	%Com
Example	33	66.0%	38	11.3%
Fact	29	58.0%	90	26.8%
Concept	12	24.0%	3	0.9%
Requests for Examples	0	0.0%	9	2.7%
Requests for Facts	0	0.0%	54	16.1%
Requests for Concepts	0	0.0%	0	0.0%
At least one of the above	45	90.0%	174	51.8%

Table 3.10: The incidence of examples, facts, and concepts in the texts of top-voted Stack Overflow answers and comments in our dataset, along with requests for these types of knowledge. The bottom row shows totals of answers and comments that discuss at least one the knowledge types described in our theory. To describe the incidence of each content type, we include: **Ans**, the number of answer texts in which it appeared (out of 50 answers); **%Ans**, the percentage of answer texts in which it appeared; **Com**, the number of comment texts in which it appeared (out of 336 comments); **%Com**, the percentage of comment texts in which it appeared.

Results

Overall, 90.0% of answers included knowledge from at least one of the three types of robust knowledge from our theory; 66.0% include examples, 58.0% include facts, and 24.0% include concepts. The balance changes for comments on these answers. For comments, 51.8% include examples, facts, or comments – or requests for additional examples, facts, or comments. Examples feature in 11.3%, facts in 26.8%, and concepts in 0.9% of comments. *Requests* for examples feature in 2.1%, requests for facts in 16.1%, and requests for concepts in 0.0%. See Table 3.10 for a summary of the numbers of answers and comments that included each knowledge or request type. See Table 3.11 for snippets of answers and comments that included each knowledge type.

Examples, with the largest incidence in the data set, also exhibited a diversity of forms. They ranged in size from single expressions to one- or two-line snippets (e.g., Example 3 in Table 3.11) to full programs (e.g., Example 1 in Table 3.11). Some users provided exhaus-

Theory Knowledge Type	Sample Content

Pattern	"...#!/usr/bin/env python3
	from functools import partial
	from itertools import repeat
	from multiprocessing import Pool, freeze_support
	
	def func(a, b):
	    return a + b
	
	def main():
	    a_args = [1,2,3]
	    second_arg = 1
	    with Pool() as pool:
	        L = pool.starmap(func, [(1, 1), (2, 1), (3, 1)])
	        M = pool.starmap(func, zip(a_args, repeat(second_arg)))
	        N = pool.map(partial(func, b=second_arg), a_args)
	        assert L == M == N
	
	if __name__=="__main__":
	    freeze_support()
	    main()..."

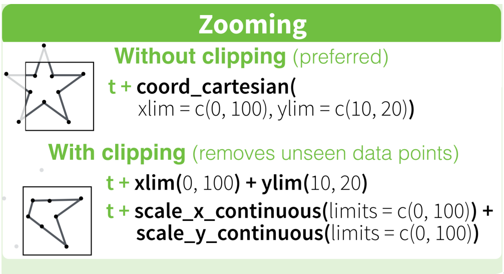
	"...// Loop through each row of the worksheet in turn for ($row = 1; $row <= $highestRow; $row++){ // Read a row of data into an array $rowData = $sheet->rangeToArray('A' . $row . ':' . $highestColumn . $row, NULL, TRUE, FALSE); // Insert row data array into your database of choice here }..."
	“@[username redacted] you can also cast context.Resouce to Microsoft.AspNet.Mvc.AuthorizationContext. Example: Microsoft.AspNet.Mvc.AuthorizationContext resource = (Microsoft.AspNet.Mvc.AuthorizationContext) context.Resource; string id = resource.RouteData.Values[”id”].ToString();”
Fact	“According to Eigen Doc, Vector is a typedef for Matrix, and the Matrix has a constructor with the following signature: Matrix (const Scalar *data) Constructs a fixed-sized matrix initialized with coefficients starting at data...”
	“...The token as given in the sample does not expire, but the handler.CreateToken takes additional parameters to either post-date a token or set an expiration date...”
	“WARNING: this is cool, but also dangerous! Because the Eigen object will NOT create its own memory. It will operate on the memory provided by ”data”. In other words, working with the Eigen object when the ”data” object is out of scope will result in a segmentation fault (or memory access violation).”
Concept	“...The async commands return immediately, while the non-async commands block...”
	“...  Zooming Without clipping (preferred) t + coord_cartesian(xlim = c(0, 100), ylim = c(10, 20)) With clipping (removes unseen data points) t + xlim(0, 100) + ylim(10, 20) t + scale_x_continuous(limits = c(0, 100)) + scale_y_continuous(limits = c(0, 100)) ...”
	“...We don’t want you writing custom authorize attributes. If you need to do that we’ve done something wrong. Instead you should be writing authorization requirements. Authorization acts upon Identities. Identities are created by authentication...”

Table 3.11: Sample examples, facts, and concepts from top-voted Stack Overflow answers and comments on top-voted answers.

tive comments (Example 2), while some provided none (Example 1). Some clearly divided code from descriptive text, while others interwove them (Example 2). Some provided examples that used special notations or descriptive variable names to make it clear where and how readers should alter examples for their own use cases (e.g., a variable named "yourClassList"), essentially providing parameterized examples. Others provided fully concrete examples, tailored to the question asker's use case or sometimes drawn directly from the commenter's own use case without adaptation. Facts also varied in form. They were often paired with either API methods (Fact 2) or API classes (Fact 1) – or sometimes example expressions – and presented as explanations of the paired snippets. Within comments, facts frequently came in the form of additional information about the behavior of examples or approaches suggested in the answer or in prior comments (Fact 3). Concepts tended to be quite briefly explained, typically by relying on other concepts that the commenter expects readers to understand (Concepts 1, 3). For example, they might explain asynchrony in terms of blocking (Concept 1) or explain an API's `apply` method by making an analogy to the host language's `apply` function. Occasionally, they presented concepts by borrowing explanatory materials from other sources (Concept 2).

In addition to the types from our theory of robust knowledge, we identified 25 other types of informational content that answer and comment writers included. See Table 3.12 for descriptions of the types of non-theory information and Table 3.13 for sample texts coded with each type. Table 3.12 also includes the incidence of each information type in the answers and comments we coded. The most common non-theory information type in answer texts was "Comparison with Alternatives," a label for information about how a particular strategy for completing a task compares with one or more alternative strategies. This information type appears in 12.0% of answer texts – a substantial portion, but smaller than the 66%, 58%, and 24% incidence rates of our robust knowledge types. Discussion of installation, the advantages and disadvantages of a particular code strategy (in isolation from a direct comparison with an alternative), and performance is also fairly common in answer texts; all of these appear in 8.0% of answer texts.

Non-Theory Knowledge Type	Description	Ans	%Ans	Com	%Com
Comparison	Comparing API or code with an alternative.	6	12.0%	12	3.6%
Installation	Details about installing or importing the target API.	4	8.0%	8	2.4%
Pros and/or Cons	Describes pro or con of a particular approach (often used with Comparison with Alternative, but some users mention a pro or con without explicitly comparing to anything else).	4	8.0%	5	1.5%
Performance	Mentions performance or shows benchmark.	4	8.0%	5	1.5%
Host Language	Information about the host language rather than the API.	3	6.0%	6	1.8%
Version Difference	Differences between versions, features that are deprecated.	2	4.0%	22	6.5%
Build Process	Discussion of how to build a project, use build tools.	2	4.0%	10	3.0%
OS	Operating sytem effects, differences.	2	4.0%	7	2.1%
IDE	Instructions for how to accomplish a task in an IDE, or comments about how something appears in an IDE.	2	4.0%	7	2.1%
Compiler	Mentions compiler effects, differences.	2	4.0%	0	0.0%
Maintained	Comments on whether an API or other resource is maintained.	1	2.0%	1	0.3%
Hardware	Mentions hardware or hardware characteristics.	1	2.0%	0	0.0%
Debugging Strategy	Strategies for how to debug; not simply debugging a particular issue, but general approaches to debugging.	1	2.0%	0	0.0%
Thanks	Thank you or compliment for an answer or earlier comment.	0	0.0%	41	12.2%
Disagreement	Disagreement with the answer or an earlier comment, pointing out a bug in earlier content, saying an approach does not work for them.	0	0.0%	25	7.4%
Agreement	Agreement with the answer or an earlier comment.	0	0.0%	13	3.9%
Code Style	Comments on style or readability of an approach, or whether code is idiomatic.	0	0.0%	12	3.6%
Goals	Discusses goals of asker, commenter, or API developer.	0	0.0%	9	2.7%
Side Discussion	Discussion about something unrelated to the question.	0	0.0%	6	1.8%
Confusion	Indicates commenter is confused or surprised, does not understand something, did not expect something.	0	0.0%	5	1.5%
Dependencies	Describes need to install or include other resources.	0	0.0%	4	1.2%
Stack Overflow Culture	Describing Stack Overflow rules or culture.	0	0.0%	4	1.2%
Unrelated Help	Requests for help with other, unrelated questions.	0	0.0%	4	1.2%
No Idea	Describes not knowing how to solve an issue, accomplish a task.	0	0.0%	1	0.3%
Complicated	Claims something is complicated, hard, annoying.	0	0.0%	1	0.3%

Table 3.12: We identified 25 types of information that are included in Stack Overflow answers and comments about API questions but are not described in our theory of robust API knowledge. Here we describe each information type and describe the incidence of the information type in our dataset of Stack Overflow answers and comments. In particular, we include: **Ans**, the number of answer texts in which it appeared (out of 50 answers); **%Ans**, the percentage of answer texts in which it appeared; **Com**, the number of comment texts in which it appeared (out of 336 comments); **%Com**, the percentage of comment texts in which it appeared.

Non-Theory Knowledge Type	**Sample Content**
Comparison	“You might want to try alternative solution presented in my answer using custom ”exclusion” annotation, that works ”like” transient modifier, but is limited to Gson parser scope.”
Installation	“Have you actually loaded kaminari by adding it to your Gemfile?”
Pros and/or Cons	“A multiprocessing con: logging library does not work well with it.”
Performance	“I am more concerned with gemms - A*A’ are noticably faster than mkl and only slighly slower than Goto. Almost 30% fast than mkl for N=1000.”
Host Language	“transient has a specific semantics in Java. You shouldn’t use it unless you want your field to really be transient.”
Version Difference	“To those for whom hjust is not behaving as described here, try theme(axis.text.x=element_text(angle = 90, vjust = 0.5)). As of ggplot2 0.9.3.1 this seems to be the solution.”
Build Process	“gradlew clean didn’t solve the problem alone for me, I had to also do as @Ozzie suggested and add the .jar to my build.gradle located in MyProject/MyProject-MyProject/build.gradle.”
OS	“I tried this way. In terminal on Windows 7 project builds perfectly but when try Run in Android Studio - have same problem...”
IDE	“In Android Studio 1.0.2, steps 2 and 3 are not needed because this line is already in the default build.gradle: compile fileTree(dir: ’libs’, include: [’*.jar’])”
Compiler	“It looks like a problem of GCC. Intel compiler gives the expected result...”
Maintained	“...The project is also quite active as you can see from their web page and that’s a very good sign as well.”
Hardware	“Typical cases where it is either useful or necessary to use a larger LDA value are when you are operating on a sub matrix from a larger dense matrix, and when hardware or algorithms offer performance advantages when storage is padded to round multiples of some optimal size”
Debugging Strategy	“In general though, I feel like it’s time I wrote a guide on how to diagnose issues like this when using the dnx (since it’s pretty different to existing .NET)...”
Thanks	“Thank you guys, this was what I need!”
Disagreement	“This is not a good answer for large scale projects. Look at the answer that uses maven instead by saneryee.”
Agreement	“Kaminari.paginate_array(...).page(1) works in my case as well”
Code Style	“Those before .getType look so non-Java to me. I’m not even able to tell what they are syntacically. Is it an empty anonymous class or what?”
Goals	“@devshorts True. Of course you could save the Take(2) result into a variable if necessary, but that wasn’t a requirement in the question.”
Side Discussion	“Ah, so funny. Now that I stalked your website I saw you wrote c# in depth, which I am reading too :P”
Confusion	“I don’t get the Add as Library option in step 2.”
Dependencies	“One nit: you need to import traceback as well.”
Stack Overflow Culture	“This is getting out of hand now. If you have questions then create them as questions, not comments.”
Unrelated Help	“@uncaught_exceptions I have similar question on Gson here(link). Wanted to see if you can help me out.”
No Idea	“ive no experience with multiprocessing on windows or even how it could possibly work as i thought windows has no fork().”
Complicated	“I have to comment that, all this is more complex than implementing a custom authorization method.”

Table 3.13: For each of the information types described in Table 3.12, the information types that are not included in our theory of robust API knowledge, we provide a text sample that was coded with the label.

In comment texts, the most common non-theory information type was “Thanks,” appearing in 12.2% of answers. Whether thank yous represent information is perhaps arguable, but since they typically suggest an answer worked for the commenter, serving as an informal endorsement and a signal of quality to other viewers, we include “Thanks” as its own information type. The second most common non-theory information type in comments is “Disagreement,” any text that indicates a commenter disagrees with knowledge described in an answer or an earlier comment, often paired with their own knowledge that conflicts with the prior description; this appears in 7.4% of comments. (“Agreement” also appears, but only in 3.9% of comments.) Comments’ third most common non-theory information type was “Version Difference,” and this category includes discussions of how changes to tools (especially to the target API, but sometimes to the host language or an IDE) affect API code; “Version difference” content appears in 6.5% of comments.

3.6 Limitations

There are several perspectives to consider limitations from. To start with, our theory itself takes a limited perspective on what tasks it considers when it defines API knowledge (for example excluding actually running the code or working in collaboration with others), which will require complementary or alternative theories with different perspectives (such as on the process of learning an API [53]). We also only considered three of many criteria to judge our theory by, but there are others that are likely be valuable. It also will take more time and the engagement of more people and perspectives (or a negative signal of their lack of engagement) to evaluate whether this theory is useful or true.

As part of our evaluation of our theory, we used two studies to test seven hypotheses based on our theory, but those seven hypotheses aren’t necessarily the best representatives of our theory, and our study designs might not be the best way to test those hypotheses. In particular, in study 1, some participants complained about unclear or irrelevant code annotations, which might mean there were problems with how we wrote and presented the annotations separate from the theory. Additionally participants mentioned a lack of time, and difficulty of tasks (particular with d3.js where many got stuck on the first task), so our task design could confound our results. Additionally, we ran many statistical tests, which increases the likelihood of us finding statistically significant results by chance. Finally, the population of Study 1 was from a large public university whose population differs in many ways from the global population of programmers or potential programmers [125]. In study 2, our sampling of StackOverflow might not have produced the best summary of StackOverflow content, and more fine-grained classifications may have produced more nuanced results.

3.7 Discussion

Among the many ways a theory can be evaluated, we chose three to focus on: unifying prior findings into a coherent model, capturing internal logic and how parts relate, and producing (and testing) falsifiable claims. We have demonstrated how our theory unifies a number of previous findings (see 3.3.3) into a coherent model consisting of domain concepts, execution facts, and API usage patterns. We believe this model has clear internal logic and clear relationships between its pieces (see 3.3.3). Finally, we have made a start at testing hypotheses generated by our theory through two studies: a user study, and a study of StackOverflow content.

In study 1, we controlled access to the three components of knowledge from our theory, and we found mixed evidence for: 1) in specific situations, each component of knowledge significantly increased task progress and perceived API understanding, 2) that when these components of knowledge are lacking, people want it, and 3) that *access* to this knowledge is not always sufficient for progress and can be overwhelming. Study 2 showed that the predominance of content on Stack Overflow is either a definition of a domain concept, a description of an API usage pattern, or a description of an execution fact about an API's runtime behavior. Much of the remaining content was not specific to one API (e.g., API comparisons or versions), information we excluded from our theory (e.g., installation, host programming language), or part of the back-and-forth communication in Stack Overflow (e.g., thanks, disagreement). Interestingly, Study 2 showed that domain concepts are rarely discussed on Stack Overflow. One possible explanation of this is that without knowing domain concepts, one cannot easily write questions or form search queries to find API documentation [31, 58, 59, 146]. Learning of domain concepts may then necessarily be learned and discovered through other media.

The mixed evidence in support of our theory, it does add nuances that demand further refinement. For example, it appears that learning concepts, facts, and patterns is not always straightforward and can be overwhelming. This is consistent with prior work showing that API documentation can quickly become overwhelming and is not always easy to find when one needs to learn it [136]. This is no different from any other formal or informal learning context, where instruction needs to be tailored to a learner's prior knowledge.

Of course, there is substantially more work to do to with this theory, both in evaluating it (more empirical studies, as well as evaluations on other criteria such as how well it aids in answering questions in the field and what questions does it generate), as well as in refining its claims. For example, our theory claims that a lack of *robust* knowledge results in task difficulties and defects, and we found mixed evidence for it. Is this true, and if so, by what mechanisms does robust API knowledge prevent defects? The theory also isn't clear on the effects of having robust knowledge of *part* of an API, but brittle or no knowledge of other parts. Are there cumulative benefits to having a broader knowledge base of robust knowledge about an API, or is the utility of knowledge highly task-specific? Finally, are there particular concepts, facts, and patterns about APIs that are more important than others to robust use of the API, or is it entirely task dependent? If so, why, and what would this mean for API documentation and tools? These unanswered questions are central to building a more powerful, predictive, and explanatory theory of API learning.

If we believe the theory, it has several implications for the design of API learning materials such as documentation, tutorials, Q&A, and classroom instruction. For example, the theory suggests that documentation should at the very least contain as many concepts, facts, and patterns known to be relevant as possible. We suspect most documentation does not, hence the popularity of sites like Stack Overflow. Second, our theory suggests that what *subset* of concepts, facts, and patterns a developer needs to know is highly-task dependent, suggesting that media like API documentation are rarely going to be structured in a way that optimally supports learning, both because they only offer one structure, and because their content is written to one level of prior knowledge. Media like tutorials, question answering sites, or even entirely new tools that attempt to retrieve and present relevant knowledge for a task and for a specific learner's prior knowledge are likely to be much more effective. Recent proposals for on-demand documentation [111], tools that have begun to automatically extract code patterns and their alternatives (e.g., [36]), and program analyses that can automatically extract execution facts about API behavior (e.g., [51], [148]), point the way to a future in which people learning an API can get exactly the knowledge they need for any given task. Our theory can help generate additional ideas for future tools, while also explaining how current tools do and do not support learning.

Until that future comes, our theory has also several implications for practice. Every day, millions of students, end-user programmers, and professional developers are encountering new APIs, trying to learn and use them productively. In the absence of great learning materials, learners try different strategies that can be related to our theory (e.g., opportunistic learners might be taking a API usage pattern first strategy, while systematic programmers might be taking a concept and fact first strategy [21, 20, 73]). By recognizing how their learning relates to our theory, learners could consider increase their awareness of the strengths and weaknesses of their strategy and make appropriate adjustments to gain all the knowledge they will end up needing or consider alternative strategies entirely.

Together, advances in practitioners' strategies for learning and in our ability to generate API learning materials that teach the knowledge that learners need, could result in a world with much lower barriers to learning a new API. In this world, not only would humanity be able to create more with APIs, but API designers would be able to change them more rapidly. We hope our theory can be a guide to the research and development needed to achieve this future.

Chapter 4

PROJECT 3: DEFINING AND EXTRACTING API USAGE TEMPLATES

4.1 Introduction

In order for programmers to successfully use application programming interfaces (APIs) they need to know many different things, such as how to search for and recognize relevant information [31, 58, 59, 146], how to set up an operating environment to run the API code [73], and how API features and other code can be combined to create the different possibilities and API enables [59, 74, 31].

In our theory of robust API knowledge (Chapter 3), we called code combinations involving APIS *API usage patterns.* We claimed developers need to know these API usage patterns in order to know how to combine code in useful ways, the rationale for those arrangements, and to know the range of what is possible to create with a given API.

Since there are an infinite number of possible API usage patterns, not all of which are likely to be useful or interesting, selecting API usage patterns becomes a search problem. Developers need to search, discover, and learn which of these infinite possible API usage patterns are useful, interesting, and relevant to their individual needs. Unfortunately, the sources of knowledge available to developers are not organized around API usage patterns, and only a few research prototypes have made steps toward organizing information in this way:

- **API feature documentation** (descriptions of function, classes, etc., e.g., JavaDocs): Explains all the parts of the API, gives developers info they can use to invent API usage patterns, though this puts the burden on the developer to do this search and invention themselves. This documentation occasionally contains some info on how API pieces can be combined (e.g., arguments from a type). Note that the auto-complete feature in many IDEs is a more convenient way of exploring API feature documentation in context.

- **Example code** (official, GitHub, gists, etc.): Example code will contain API usage patterns in specific concrete usages (which aids developers in creating runable code). The concreteness of examples is in conflict with the generalization needed for exploration. For example, you can run the ThreeJS code "`new THREE.BoxGeometry( 1, 3, 5 );`" but it will have no effect unless you also create and set up a scene which you add the box to, but that extra code setting up scene may distract a developer from learning about creating boxes. Also the more generalized and informative version of that code "`new THREE.BoxGeometry(«width», «height», «depth»);`" won't compile until width, height and depth are filled in. Previous research has found that examples need to be scaled to the right size (just the relevant API usage pattern and as little extra as possible) to be understood efficiently [110], though we suggest that removing the extra context to make code concrete and runnable can be valuable to developers. Examples are also generally disconnected from each other, making them hartod toto explore. Relevant examples are difficult to find and to adapt [146, 119, 106, 30]. And "one type of example that seemed to be consistently missing were examples showing code to integrate multiple APIs" [110].

- **Tutorials** (official or e.g., blog posts): Tutorials often involve examples with descriptions of how to build them in separate steps. By showing related examples (step by step changes) they imply ways the API usage patterns used to change the examples. Additionally, tutorials sometimes write out API usage patterns separately like we suggest. Still, tutorials are linear, thus only showing API usage patterns and possibilities surrounding a single use case.

- **Crowdsourced sources** (e.g., StackOverflow, discussion forums): Responses may include examples or even a small API usage patterns. Still, information is organized by answering questions. When a question aligns with what a developer wants to do, these sources may provide the information the developer needs. But if no question aligns enough with their situation (or they aren't able to find the right question), these sources wont help. Additionally, the organization by question makes it hard to explore other possible API usage patterns.

- **Code generators** (e.g., Ruby on Rails' "rails generate"): These create code (whether initial project or additions to a project) that often contain multiple calls to API. While useful, these only incorporate a small number of preset API usage patterns.

- Oney and Brandt created an interface for searching for and inserting *codelets*, which are manually created code templates with parameters that can be filled in [89]. While these do encode API usage patterns, they are limited by needing to be manually created and they are not interconnected to each other in the way we will suggest in this paper.

All the above sources depend on human's directly specifying the API usage patterns including specifying how it can be parameterized. The resulting sources therefore either have data that either doesn't show how example code can be parameterized, or that shows only a very small set of API usage patterns. To get larger and more useful sets requires automatically processing of larger corpuses of code and examples. Some research prototypes have done versions of this:

- Many papers have tried to infer API usage patterns or API usage rules, focusing on unordered patterns, ordered patterns, behavioral specifications, migration mappings, and other properties like control structures [107].

- Glassman, et al. visualized code examples by combining them into a common (and predefined) structure [37]. This predefined structure acted as parameterized points so

Viewing and selecting along this structure showed developers different ways code could be filled in.

- Clone detection algorithms use various mechanisms to identify code similarity by searching for similar lines of code, program metrics, AST subrees, or dependency graphs [10, 114].

We propose creating a data structure represents an interconnected set of parameterized API usage patterns with metadata describing their purpose. Unlike the prior automated methods, our data structure preserves the code structure with minimal assumptions about that structure. For example, figure 4.1 shows an API usage template with a structure that cannot be captured with simple sets ordered or unordered API calls. We believe that by making this data structure align with developer needs to learn and understand API usage patterns, it will be better support developer needs in regards to API usage patterns and be an important and beneficial complement to other available resources.

To represent API usage patterns, we will store the patterns as individual *API templates*, each consisting of a generalized abstract syntax tree (G-AST) with metadata. The G-ASTs will represent the structured API usage pattern code as a traditional abstract syntax tree but with some nodes being parameterized. A piece of example code will be said to use an API Template if the G-AST appears in the code's AST as a graph minor embedding. The metadata included with each template consists of: explanations of the purpose of the template, which templates are subtemplates of others (that is when one template's G-AST used in another one's), and information about the prevalence of templates (e.g., what percentage of GitHub projects with the API use this template). Since there are an infinite set of potential G-ASTs for any given API, we need to restrict it to just the templates that are in some way interesting and useful, as well as different enough from other templates (there could be a combinatorial explosion of small variations of templates). We will operationalize "useful and interesting" later in our algorithm by using prevalence of templates.

We next develop an algorithm to produce G-ASTs for this data structure from example

```
// Create an interactive map that lets users select by drawing boxes

// Create an interactive map with a vector tile layer
const map = new Map({
    target: 'map',
    view: new View({
        center: [0, 0],
        zoom: «...»
    }),
    layers: [
        new VectorTileLayer({
            source: new VectorTileSource({
                format: «...»,
                url: «...»
            })
        })
    ]
});

// add a DragBox interaction used to select features by drawing boxes
const «?1» = new DragBox({
    condition: platformModifierKeyOnly
});
map.addInteraction(«?1»);
```

Figure 4.1: Example API usage template for the interactive mapping library OpenLayers, based on their official examples [4]. The comments describe the purpose of the code. The `«...»` locations must be filled in with some content. The `«?1»` is any identifier name. like `dragbox`, to hold the `DragBox` interaction object and add it to the map.

code found on gist.github.com and evaluate the algorithm's output by having users familiar with the API we tested it on evaluate the results. We then demonstrate the benefit of API usage templates by proposing how our data structure can aid users in 12 different scenarios (section 4.6). Finally, we place developers in one of those scenarios (Scenario 1: "I want to learn what an API can do and how to do it") and study the effects of providing templates based on our automatically generated G-ASTs.

4.2 Related Work

Automatically generating documentation and tutorials to aid programmers while working with APIs has been a goal for the research community [112, 129]. To this end various projects have provided help by linking developers to resources and synthesizing those resources for developers.

4.2.1 Linking developers to resources

Some projects have helped developers search for official or unofficial documentation, sometimes in an Integrated Development Environment (IDE) [95, 96, 134, 66, 109]. Other projects help connect parts of the code a developer is working on to documentation [127, 50, 49]. Jiang, et al., automatically found input/output examples for functions [51]. Robillard created a system that, given some code elements in program, automatically found others that might be interesting to a software developer [108]. The most relevant of these to our work are systems that help link developers to code examples [149, 56, 25]. Unlike these methods, we focus on the synthesized usage patterns we extract (though we can still use our data structure to link back to examples as we explain in section 4.6, scenario 7).

4.2.2 Synthesizing resources

Other work synthesized resources for developers. Ribillard et. al. has a literature review of various methods of inferring API properties, such as usage rules and patterns, though the patterns found are in the forms of unordered or ordered sets [107]. Some work has sought to capture more structure than this, such as info about control structures [87] or by fitting examples into a common, predefined structure [37]. Our data structure is capable of capturing much more diverse structures in code (like in figure 4.1).

A separate set of relevant work is that in synthesizing code explanations [104, 94, 137, 24], and providing them to users [42]. Automatically incorporating code explanations is an important next step that is out of scope of this paper.

4.2.3 Code templates and code generation

Some work has focused on helping developers write code in their IDE. Oney and Brandt created an interface for inserting *codelets*, manually created parameterized code templates with descriptions [89]. Some take API usage patterns in the form of ordered sets and use those to generate examples [86] or suggest code changes [113]. Our suggested data structure should support this type of work as well (e.g., scenario 9 in section 4.6), but with the benefit of our more flexible ability to represent code structures.

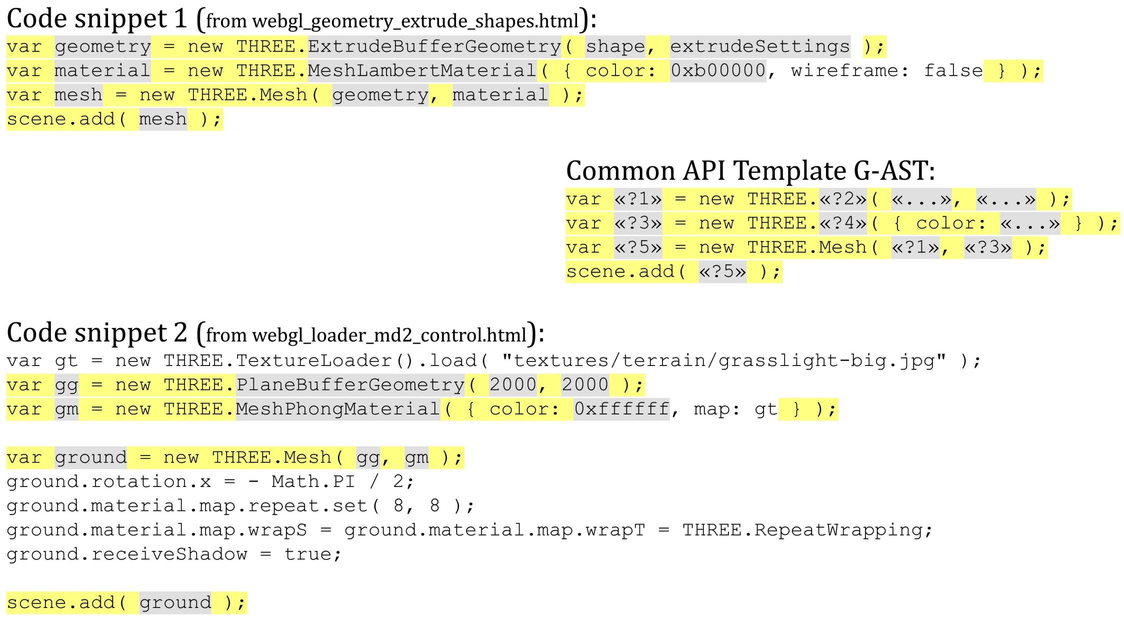
4.3 The API Usage Template Data Structure

The first contribution we make is a definition of a data structure that represents API usage patterns, which we call *API Usage Templates* or *templates* for short.

Templates are at their core example code with metadata. The metadata will encode information on parameterization and explanations, as well as how templates relate to each other.

Representation of code in templates

There are two perspectives on code that we want to keep track of when we put templates into data structures: the underlying structure of the code (the compiler's perspective), and the way code is displayed and formatted (the human developer's perspective).



Code snippet 1 (from webgl_geometry_extrude_shapes.html):

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Common API Template G-AST:

```
var «?1» = new THREE.«?2»( «...», «...» );
var «?3» = new THREE.«?4»( { color: «...» } );
var «?5» = new THREE.Mesh( «?1», «?3» );
scene.add( «?5» );
```

Code snippet 2 (from webgl_loader_md2_control.html):

```
var gt = new THREE.TextureLoader().load( "textures/terrain/grasslight-big.jpg" );
var gg = new THREE.PlaneBufferGeometry( 2000, 2000 );
var gm = new THREE.MeshPhongMaterial( { color: 0xffffff, map: gt } );

var ground = new THREE.Mesh( gg, gm );
ground.rotation.x = - Math.PI / 2;
ground.material.map.repeat.set( 8, 8 );
ground.material.map.wrapS = ground.material.map.wrapT = THREE.RepeatWrapping;
ground.receiveShadow = true;

scene.add( ground );
```

Figure 4.2: Example API Usage Template G-AST that is shared by two pieces of official example code for the 3D Graphics library ThreeJS [80]. The purpose of the shared code is to create a 3D object (a mesh composed of a geometry and a material), and add it to a scene. Shared AST nodes are highlighted in yellow, and shared generic nodes are highlighted in gray.

In order to represent the underlying structure of the code, each template will have an abstract syntax tree (AST)[1] of the code, which we think is the most natural compiler perspective of the code. Since we want our templates to be parameterized and relate to other templates, we want to allow gaps and unspecified details in our code, we will make two new generic types of nodes in the ASTs. We will call these ASTs with generic nodes *Generalized Abstract Syntax Trees* or *G-ASTs* (Fig. 4.2). The first generic node type will be *generic nodes*, which are placeholder nodes that we match with any PL-valid node or subtree. We add a second generic node type for *generic names*, which will represent a string reference (like a variable name or object field), which can be matched to other identical identifiers throughout the code, but the string contents could be anything. This second type of node allows us to, for example, trace variable use while allowing different programs to choose different variable names.

Since these G-ASTs are intended to represent a generic version of code, we need to know when the G-AST is contained in another G-AST or in a piece of code (AST). We will define containment using *graph minor containment*. Graph A is said to have a graph minor containment in Graph B if Graph B can be modified by deleting nodes and contracting the edges in such a way to produce a graph isomorphic to A [105]. With our G-AST, which are ordered trees, we use the graph minor containment (with the additional allowances for matching generalized nodes to more specific nodes, and sorting child nodes that are unordered, like keys in object or dictionary constructors) to determine whether a G-AST is contained in a code sample or another G-AST (see Fig. 4.3 and Fig. 4.2). Therefore, an example can contain a G-AST with additional nodes inserted at any point in the G-AST (like the addition of "2 *" in Fig. 4.3). If we were to use subtree containment, then we would need to have generic nodes for every place code might be added to a G-AST, but where nothing could be added, which we feel could make the G-ASTs unreadable, and also didn't seem to us to match our intuition about what generic code should look like (for example,

[1] We assume every file or program has a root node that contains all the ASTs of that file or program.

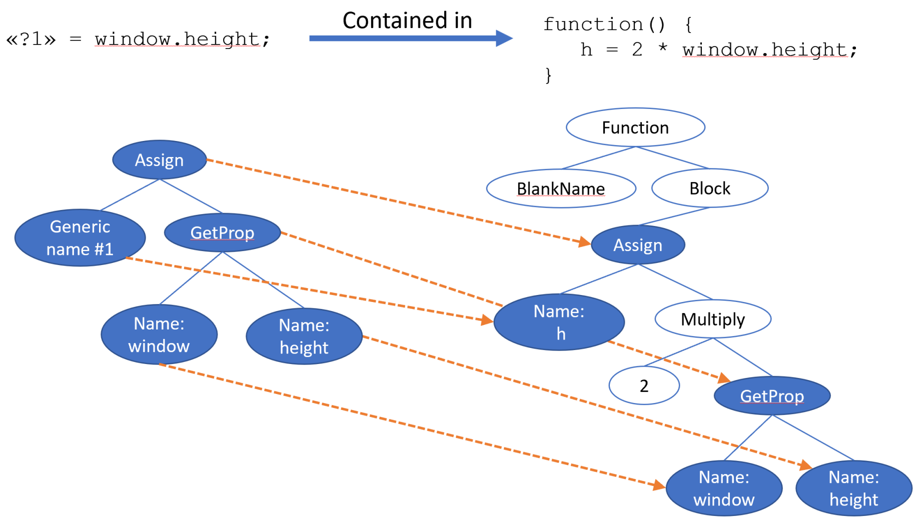
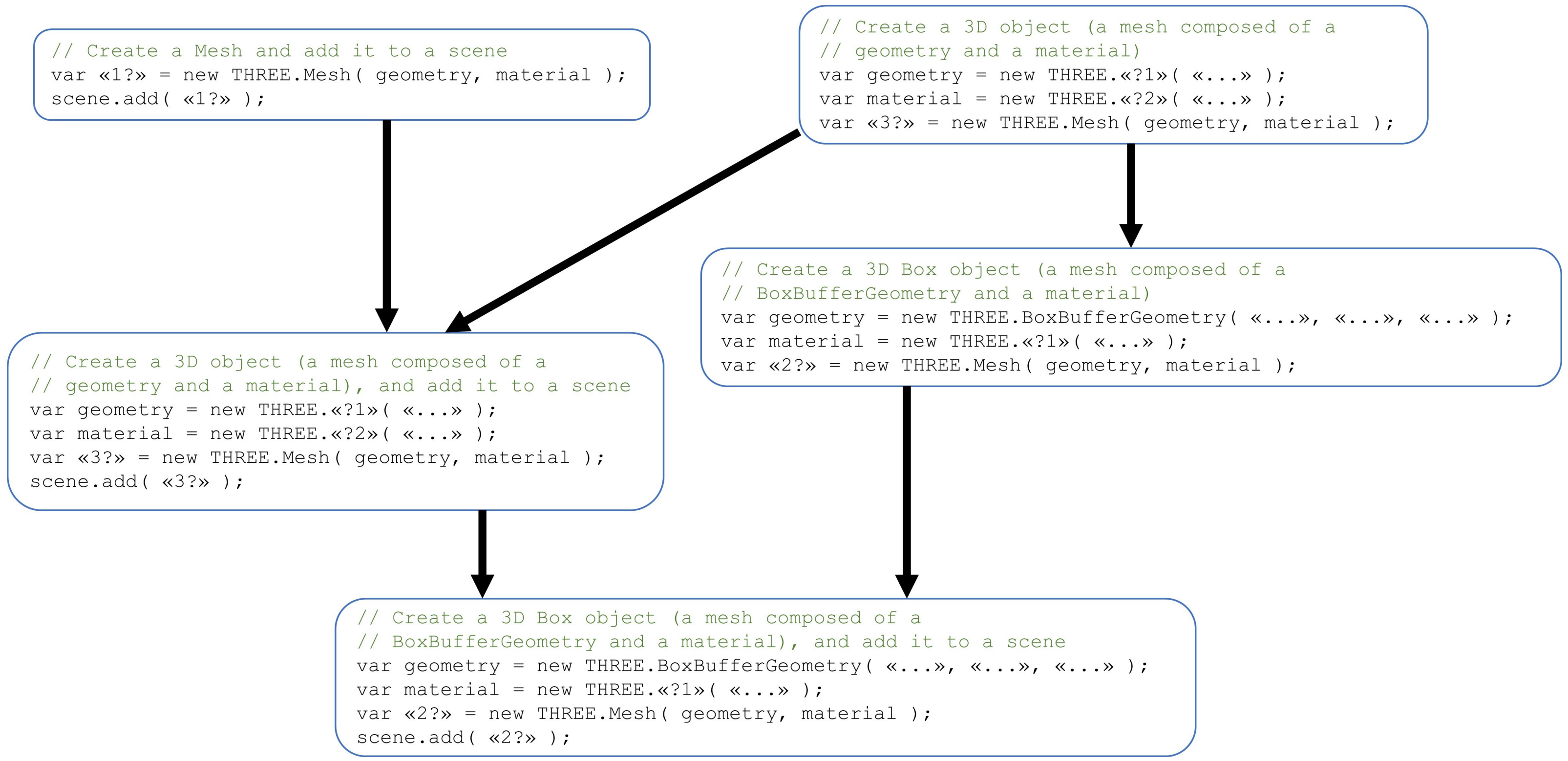


Figure 4.3: An example of G-AST containment. The G-AST on the left is contained in the G-AST on the right. Both are represented in code structure and tree structure. The red arrows show a mapping from the first G-AST to the G-AST it is contained in.

see the extra unmatched code in the snippets in Figure 4.2). Graph minor allows G-AST to capture overarching patterns which can be customized in many different ways in specific examples.

Now that we have defined code containment, we can define a relationship graph of many G-ASTs (Fig. 4.4. In this graph, edges represent G-AST containment (we can optionally remove containment edges which are implied transitively, for example if A is contained in B and B is contained in C, we don't need a separate edge to tell us A is contained in C). We will call a template whose G-AST is contained in another template a *subtemplate* of that other template. This graph then can be used to find parameterization information from any G-AST by looking at what all of the G-ASTs that contain the original G-AST put in any spot.



```
// Create a Mesh and add it to a scene
var «1?» = new THREE.Mesh( geometry, material );
scene.add( «1?» );
```

```
// Create a 3D object (a mesh composed of a
// geometry and a material)
var geometry = new THREE.«?1»( «...» );
var material = new THREE.«?2»( «...» );
var «3?» = new THREE.Mesh( geometry, material );
```

```
// Create a 3D Box object (a mesh composed of a
// BoxBufferGeometry and a material)
var geometry = new THREE.BoxBufferGeometry( «...», «...», «...» );
var material = new THREE.«?1»( «...» );
var «2?» = new THREE.Mesh( geometry, material );
```

```
// Create a 3D object (a mesh composed of a
// geometry and a material), and add it to a scene
var geometry = new THREE.«?1»( «...» );
var material = new THREE.«?2»( «...» );
var «3?» = new THREE.Mesh( geometry, material );
scene.add( «3?» );
```

```
// Create a 3D Box object (a mesh composed of a
// BoxBufferGeometry and a material), and add it to a scene
var geometry = new THREE.BoxBufferGeometry( «...», «...», «...» );
var material = new THREE.«?1»( «...» );
var «2?» = new THREE.Mesh( geometry, material );
scene.add( «2?» );
```

Figure 4.4: A relationship graph of multiple API usage templates for the 3D graphics library ThreeJS. The black arrows point from a subtemplate to a larger template that contains the subtemplate.

Representation of explanations in templates

There are two different types of explanations in templates that we have to make sure are represented: summary of what a section of code does, and the purpose a section of code has in relation to the larger template. In order to store summaries of what each section of code does, we will have each template include a text *code summary field* that each template has. Since templates may also have links to subtemplates, we will have access to *code summary fields* in the subtemplates. In order to store the purpose of sections of code in relation to the larger template (which may be different than the *code summary field* of the subtemplate), we will add *purpose nodes* to the AST, which will apply to its child nodes.

Other metadata in templates

In addition to the template metadata we have already included in our templates, we will also suggest storing the *prevalence* of the template in a corpus of code examples. This will allow us to privilege prevalent options when deciding what templates are worth storing in our system and worth displaying to users.[2] There may be other valuable pieces of metadata to include as well, such as links to the examples a template is found in.

4.4 Algorithm for deriving G-ASTs for Templates

Now that we have defined the API usage template data structure, we describe an algorithm that derives part of this structure: the G-ASTs. This is a challenging enough task on its own that we leave deriving explanations for future work. Our algorithm takes a corpus of programs (we limited ourselves to JavaScript, though we believe our algorithm to be generalizable), and outputs a list of G-ASTs.

[2] When computing prevalence, we will to control for situations where identical code has been copied, for example if a file using an API has been copied into many projects.

4.4.1 Parse ASTs

The first step of our algorithm is to parse the plain text corpus of programs into ASTs (each program had a root node which contained all the ASTs for the program, this way each program is a single tree, which simplifies how we store and compare them later).

4.4.2 Finding shared G-ASTs in code sets

Next, we find G-ASTs that are shared between multiple files because each file has a combinatorially large possible set of G-ASTs, and shared G-ASTs guarantee that a prevalence of more than one file. To find shared G-ASTs, our algorithm uses chooses code sets, which consist of one source file AST and a set of reference file ASTs (in practice we found that having a single reference file AST seemed to work well, and when having more, it took many more code sets to find more than just the most basic patterns). Our algorithm then builds a G-AST piece by piece based on the source file, at each step verifying that the constructed G-AST appears in all the reference file ASTs. Since the number of possible code sets (source files plus reference sets) can be enormous, we did not find it feasible for computation. We therefore generated many randomly selected code sets to build G-ASTs from.

There are still what we find to be an infeasibly large potential set of shared G-ASTs in our code sets, so we take two additional steps to select fewer and more meaningful G-ASTs: Using data and control dependencies, and selecting the largest shared G-ASTs.

4.4.3 Find Data and Control Dependencies

One aspect that we consider makes a G-AST meaninful is that it consists of code that works together to achieve some purpose. Though our algorithm does not know the purpose of the code it is looking at, it can determine what code is working together by tracing data and control dependencies [34]. We take the following steps to find the data and control dependencies:

Our algorithm finds data and control dependencies in the source file AST. To find these, we recursively go through the AST (while generating a variable table for later use), and make a control dependency graph and a data dependency graph with links to represent the dependencies. For control dependencies, any node that appears inside a control structure (e.g., for loop, function definition, etc.) depends on that control structure. For data dependencies, we track the flow of information, one step at a time, representing those with links. For example, in the line of code “a = 2 + b” the “a” node depends on the assignment node, which in turn depends on the addition node, which in turn depends on the 2 node and the b node. The b node then depends on wherever b was last assigned (we use the variable table to track this). Similarly, function calls depend on their arguments, if statements depend on the conditional argument, etc.[3] We also use our variable table to track which nodes are references to something not defined in the file, which we observed tend to be references to the API as well as built in JavaScript objects like “window,” and “console.” In code snippet 1 in figure 4.2, “THREE” is an external reference to the API, and all other identifiers refer to something defined in the file itself (though some not part of the displayed snippet).

4.4.4 Find Largest Shared G-ASTs

We now use the dependency graph that we build from the source file AST to start building G-ASTs whose nodes are all related. We start with a node that represents an external dependencies (which we expect to be API references).[4] We then recursively add nodes from forward or backward data dependencies or backward control dependencies,[5]. This process creates G-ASTs of related nodes that we therefore hope have a common purpose and would

[3] We did not include script nodes or expressions (defining a line of code, usually with a semicolon at the end), as they were not necessary to track the data flow, though it might improve the final G-ASTs since that would add back in the semicolons to many lines.

[4] In a language like JavaScript that has built in objects that can be referenced, like “window” or “console,” these should be not be considered.

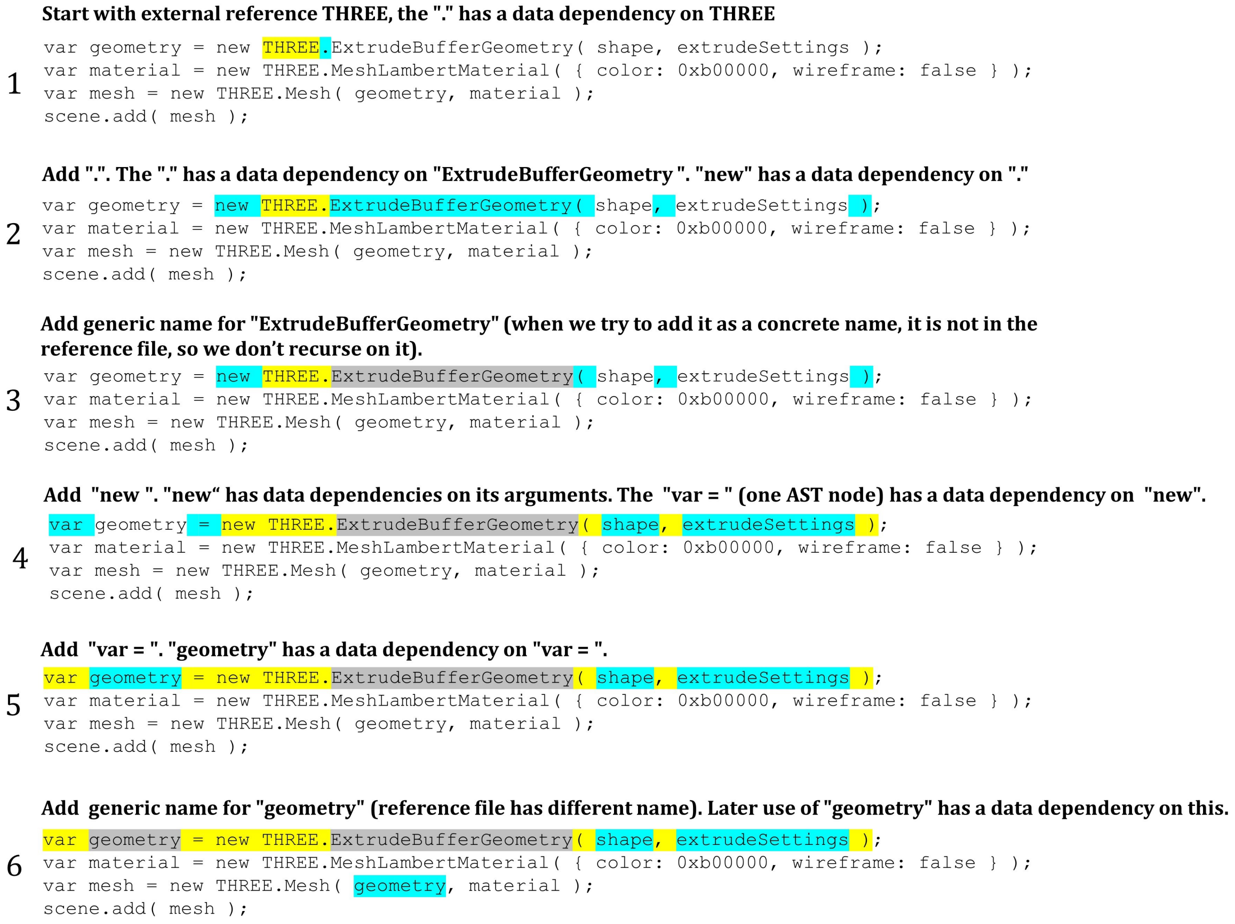
[5] Following control dependencies forward provides too many options, especially if we follow control dependencies backwards (e.g., every node in a function has a control dependency on the function, and therefore if the function was included we would try adding any combination of nodes, which may be less related than we want.

make good candidates for meaningful API usage templates.

As we do this recursive process of adding nodes, we check with every node that we add that the G-AST is still contained in all the reference file ASTs. When we can add no more nodes (a local maxima), then we stop and label the current G-AST as a *largest shared G-AST*. When we have finished all recursive options, we have a set of *largest shared G-AST*, which we define as a G-AST from a source file that is present in the reference set, but which no more nodes can be added to from forward or backward data dependencies or backward control dependencies, with it still being in the reference set (the common G-AST in Fig. 4.2 is the largest shared G-AST of the two code snippets). The reason that we only save the largest shared G-ASTs and not all the G-ASTs that we constructed along the way is that this both reduces the enormous set of potential G-ASTs we will find, but it also selects what we hope are the most meaningful G-ASTs, because we expect code to be used in natural chunks, and that the differences in code between what different files use will be along these natural chunks.

Our full algorithm, therefore, is as follows: create a set of the *selected nodes* at first containing only add only the external reference node ("THREE" in Fig. 4.5 step 1). Now, recurse along the edges of the data and control dependency graph as follows:

1. Generate the G-AST from the *selected nodes* set. This is done by taking the source AST and removing all the nodes that are not in the *selected nodes* set with three of caveats to preserve code structure and semantic rules (otherwise the G-AST looks "wrong" and it won't be clear how to fix it): 1) If a deleted node is a child of a selected node where child place number matters (e.g., left hand of an assign, or 3rd parameter of a function), then replace the deleted node with a placeholder generic node. 2) If a block (e.g., curly braces in JavaScript, indentation in Python) has a parent that is included, keep the block since the parent structure needs it to make sense (e.g., a for loop). 3) If a generic node has multiple active children, keep the node, as a placeholder of the structure of the code.



Start with external reference THREE, the "." has a data dependency on THREE

1

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Add ".". The "." has a data dependency on "ExtrudeBufferGeometry ". "new" has a data dependency on "."

2

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Add generic name for "ExtrudeBufferGeometry" (when we try to add it as a concrete name, it is not in the reference file, so we don't recurse on it).

3

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Add "new ". "new" has data dependencies on its arguments. The "var = " (one AST node) has a data dependency on "new".

4

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Add "var = ". "geometry" has a data dependency on "var = ".

5

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Add generic name for "geometry" (reference file has different name). Later use of "geometry" has a data dependency on this.

6

```
var geometry = new THREE.ExtrudeBufferGeometry( shape, extrudeSettings );
var material = new THREE.MeshLambertMaterial( { color: 0xb00000, wireframe: false } );
var mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );
```

Figure 4.5: The first six steps of building a largest shared G-AST based on the code in figure 4.2, with code snippet 1 as the source and code snippet 2 as the reference. The yellow highlight represents selected nodes. Grey highlights represent nodes selected as generic nodes. The blue highlight represents nodes that are potential candidate nodes for recursion (forward and backward data dependencies and backward control dependencies (though there are no control dependencies in this example).

2. Check if the G-AST is contained in the all the reference ASTs.[6] If the G-AST is not contained in the reference AST, we return. For example, in Fig. 4.5, in step 3 when ExtrudeBufferGeometry was added as a generic name, it succeeded, but if it was added as a full name node, the recursive step fails and we return from that step.

3. Recurse once for each of the possible additions to the *selected nodes* set. The options for possible addition are any the forward and backward in data dependencies of any *selected node*, and backward in control dependencies of any *selected node*.

4. If none of the recursive calls had G-ASTs that were found in the reference set, then the current **selected nodes** set generates a largest shared G-AST, which we save in a set of the largest shared G-ASTs.

To speed up the algorithm and prevent duplicate work we do one further action: at each recursive step we track if the *selected nodes* set has already been seen before (the algorithm tracks the set of all *selected node* sets to do this). If we have already seen that *selected nodes* set before, we return from this recursive case doing nothing.

After completing this, we have a set of largest shared G-ASTs from this code set. We also add to this list subtrees of each block in all the G-ASTs (e.g., the contents of functions), since, on experimentation, we noticed that these are an additional set of potentially meaningful groupings of code in each G-AST.

[6] To do this we use the tree inclusion algorithm from Kilpelainen and Mannila [55], which we modified as follows to fit our G-AST containment definition: We allow generic nodes to match anything. For object definitions, we sort by key values alphabetically before checking, since they could be ordered in different ways and still be essentially the same code. To check generic name matches, we first modify their algorithm to recover the left mapping from the e matrix. Given a mapping, we check if generic names that were the same map to nodes that have the same name. If they don't match we recurse with that name match set and with it disallowed until we can't find any more mappings, or we find a mapping that preserves the generic name sets.

4.4.5 Trim and Filter G-ASTs

After finding the set of largest shared G-ASTs, we noticed that many subtrees of the G-ASTs looked too generic when testing it with the OpenLayers and ThreeJS JavaScript libraries, so we added a process to cleaning them up the G-ASTs. What we mean by subtrees that are "too gneneirc" are subtrees with name nodes that are only or mostly generic names that we felt didn't represent any concepts in how an API was used. For example, the following G-AST produced by the algorithm off of d3.js code (before this trim and filter step) has definitions of a function and calls to it and other functions with all generics, and we believe most of the G-AST has no clear purpose:

```
  var «?1» = «...»;
  var «?2» = «...»;
  var svg = d3.select(«...»).append("svg").attr("width", «?1»).attr("height", «?2»);
  function «?3»(«?4»)  {
    «?6»[«?4»]
  }

  «?3»(«...»)
  «?7».«?8»(«...»).«?5»(«...», function ()  {
    «?3»(«?9»)
  }
)
```

In order to clean this up, we do two trim passes and a filter pass, all based on heuristic experimentation with the largest shared G-ASTs found using code examples from the OpenLayers and ThreeJS libraries.

The trim passes consists of removing subtrees that our algorithm deems to be too generic. For the first pass, we define nodes that are not generic nodes, generic name, or blocks to be *concrete children.* Then recursively, from leaves and going up, we check and see if the nodes that we decided require concrete children do in fact have concrete children, and if not we delete them. We created the categories of rules for children requirements (based on what common code patterns we saw in G-ASTs that seemed to us unmeaningful):

- Any assignment node (e.g., ”=”, ”+=”, etc.) we decided requires at least one concrete right child.

- Some nodes we decided require at least one concrete descendant, such as Scripts, function calls, math operations, switches, returns, for, while, expressions, functions, object get props

- For chained accessors (e.g., object get props) we additionally remove extra “get props” (dot operator) that don’t add anything

- For object or dictionary constructors which have unordered children, we decided to require its key to be concrete so it can be sorted when determining containment (since object children can be in any order, but our containment algorithm depends on being able to sort object children, which we can’t do with generic nodes or names). If they key is not concrete, we delete the node and all its descendants.

The second trim pass (which was added after our expert review in 4.5.2) we did to remove outer functions definitions (at the top of the tree) that are too generic. Again, this was based on our observations about patterns we saw in G-ASTs that we didn’t represent anything meaningful about using the API. We did this by removing any top-level functions definitions that have generic names and all the parameters it takes are also generic.

After the two trim passes, the previous G-AST example looks like this:

```
var svg = d3.select(«...»).append("svg").attr("width", «?2»).attr("height", «?1»);
  «...»(«...», function ()  {
  }
)
```

Note that the anonymous function on the second line is kept because we counted the blank name of the anonymous function as concrete (we wanted to make sure we didn’t lose meaningful anonymous functions, but perhaps blank names shouldn’t be counted as concrete).

The generic function that takes the anonymous function as a parameter was kept because is the anonymous function counted as a meaningful argument.

Even after the two trimming passes, we noticed some G-ASTs still had so many generics and so few concrete names that we considered their purpose unclear. We therefore decided to add a filtering step that filters as follows: For each G-AST, we counted the number of generic nodes and generic name nodes, and then we count the number of concrete nodes (names, numbers, strings, “true”, “this”, etc.)[7]. Based on experimentation and observations of what G-ASTs we thought had clear purpose, we defined the *low information ratio* as (generic nodes + generic names) / (generic nodes + generic names + concrete nodes) and removed any G-AST with a low information ratio of less than 0.5. For example, the above example, post trim passes, has an information ratio of 0.89, so we kept it.

4.5 Preliminary Evaluation of G-ASTs

In order to evaluate the effectiveness of our algorithm in finding API usage patterns, we chose to evaluate how well it extracted G-ASTs from the JavaScript library d3.js (or “d3”) [12], a popular library used to create interactive data visualizations. From our prior experience learning and using d3, we knew example code was readily available online, and we also knew where to find developers familiar with it.

4.5.1 Generating d3 G-ASTs

To find a corpus of d3 code examples, we scraped github.com’s gists (gists are often small self-contained examples) for the inclusion of the string “src=https://d3js.org/d3.v5” (d3 version 5 included in a way that means references to it should start with “d3.”) and with JavaScript and HTML files. We found 466 projects. We then went through all the projects and copied

[7] We don’t include in either count generic nodes as function call parameters and generic nodes with multiple children since these nodes act as placeholders maintaining the code structure, and we neither wanted to count as meaningful or unmeaningful. We also ignore the reference to the API object (e.g., “d3”), since we assume that to be a part of the API template, and we again didn’t want to count it as meaningful or unmeaningful.

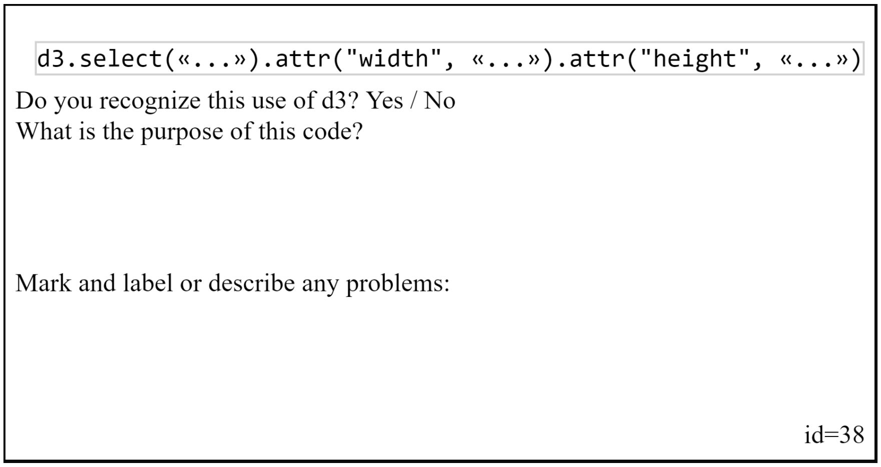
all the JavaScript code from .js files and from <script> tags in html files, and put them into a single .js file. While doing this, if one of the js files or html <script> tag was identical to code we had processed before, we did not include it (since exact duplicates could cause problems when we find the largest shared G-ASTs). This resulted in 400 JavaScript files. Finally, we loaded all of these files into our algorithm, skipping ones that did not parse properly due to limitations in the JavaScript parser we used (e.g., our parser could not handle imports or string templates). In the end we generated ASTs for 219 files (one AST per file).

To generate API usage pattern G-ASTs for d3, we had our algorithm chose 200 random code sets (one source file AST and one reference file AST) and then find, trim, and filter the largest shared G-ASTs. Our algorithm output 269 G-ASTs.

4.5.2 Expert Review

In order to judge whether the API usage pattern G-ASTs that we found were valid and whether there were patterns that we could not find, we found two participants familiar with d3 to evaluate the G-ASTs we found (to evaluate precision) and supply us with additional patterns to see what our algorithm missed (to evaluate recall).

We recruited among PhD students in the University of Washington that had used d3 as part of their research and coursework and found two participants willing to participate in our study. After reading and signing a consent form, we had them write down information about their background in response to the questions: “Tell me about your experience with d3? How did you learn d3? What have you made with d3? What parts of d3 do you know? What are you still learning?” We then had them look at an example APU usage pattern G-ASTs (displayed in the JavaScript fashion) from OpenLayers to explain how to read the patterns (including generic nodes and generic names). We then gave the a stack of printed cards (an example is in Fig. 4.6) with G-ASTs and questions. The G-ASTs were in a random order (same random order for each participant), and gave them about 40 minutes to answer these questions on as many of the cards as they could. Finally we gave them around 10 minutes to write down any API usage patterns they could think of that they hadn’t seen in



```
d3.select(«...»).attr("width", «...»).attr("height", «...»)
```

Do you recognize this use of d3? Yes / No
What is the purpose of this code?

Mark and label or describe any problems:

id=38

Figure 4.6: Example card with a APU Usage Pattern G-AST for participants to evaluate.

the cards.

After the expert review was complete, one researcher did open coding on their written answers to find any mentions of problems or questions. We did not feel there was a need for independent verification by another researcher since we are focusing on describing the categories and not on quantitative analysis (see full dataset with codes in Appendix 4.5.2).

4.5.3 Results

P1 filled out 26 cards and P2 filled out 40 cards. The G-ASTs, participant answers, and our labels are all in Appendix 4.5.2.

P1 marked that they recognized all 26 API usage pattern G-ASTs they looked at, while P2 marked that they recognized 29, didn't recognized 10 (ids 6, 7, 11, 12, 17, 21, 24, 27, 30, 39), and they left the first one blank. Of the 26 G-ASTs they both looked at, both wrote a description of the purpose of 18 (70%), P1 alone described the purpose of another 4, and speculated on the purpose of another one (21), P2 described alone one, and there were two G-ASTs that neither were able to describe. Of the 14 that only P2 looked at, P2 described the purpose of 9 (64%), speculated on the purpose of one (id-33).

We identified the following categories of problems:

- **Too abstract**: The most common complaint about our G-ASTs was that they were too abstract, with about half having some version of that problem. We therefore broke this down into further subcategories:

  - **Overall too abstract** (ids: 6, 11, 27, 30, 36, 39): These were G-ASTs where there were too participants could not describe the code's purpose. It appears that these all had generic nodes in places that were necessary to make meaning of the code (e.g., G-AST 39: "`d3.«?1»(y)`").
  - **Function definition too abstract** (ids: 5, 14, 16, 24, 26, 40): While most of these G-ASTs had sections of code that the participants described the purpose of, they also had function definitions that participants considered too abstract. The functions either had generic nodes or were anonymous functions. Sometimes the function was the parent node but it was unclear why the rest of the code was in a function. Sometimes there was a separate function definition after the rest of the code, but again with unclear meaning.
  - **Function call too abstract** (ids: 1, 14, 15, 18, 24, 33, 34, 40): As with the last category with function definitions, participants were generally described the purpose of parts of the code, but there were function calls with generic nodes that they considered too abstract (e.g., the second line of G-AST 24: "`«?2»(data)`"). These function calls often were separate lines of code from the code that participants did describe.

- **Incorrect** (ids: 3, 11, 21, 27, 31). These were G-ASTs participants labeled as having code that was incorrect. G-AST 3 was marked for not making '.range' a function call (when we checked the source and reference files for this G-AST, it was indeed always a function call, meaning this is probably a bug in our algorithm implementation). G-AST-11 was marked for passing x to a function and then calling x as a function (this looked was in fact how the source file used it, though in the G-AST match in the reference file matched unrelated “x” identifiers). Finally G-ASTs 21, 27 and 31 all had lines of code that were just a variable name with no actions.

- **Disconnected** (ids: 1, 18, 21, 33): These G-ASTs were marked as having different parts of code that were not related to each other. These problems seem to come from generalized nodes. Perhaps with is from our algorithm, so, for example, in G-AST 21, the width and height identifiers might match the «?1» and «?2» later in the G-AST.

- **Incomplete** (ids: 1, 2, 8, 19)”:In G-ASTs 1 and 2, P2 noted that some generic names only appeared once so were either not used after being saved to, or not defined before being used. G-ASTs 8 and 19 both selected a d3 element and saved it into a variable without doing anything with it after. In all of these cases, we think the G-ASTs might still be valuable as a component that is part of a more complete program.

- **Outer brackets** (id: 30): One additional problem found by P2 was in G-AST 30, which had brackets around the whole AST that P2 said was not meaningful. This might represent another bug in our algorithm implementation.

We additionally identified two more categories that were potentially relevant:

- **Speculations or Questions** (ids: 4, 17, 19, 21, 22, 26, 29): Sometimes the participants wrote down speculations or questions. Most of these were speculating at what concrete nodes would replace generic nodes (19, 21, 22, 26, 29). We were able to check these speculations and most of them looked to us like legitimate uses in the source code file, but sometimes in the reference code file the code had a different meaning. For example, in G-AST 17 has “`data`” with a function called on it that takes a parameter “`d`” in line 2 (“`data.«?2»(function d(){`”, though we think there is a bug that displayed “d” outside the parentheses), which fits well with the source file where a variable “`data`” has the function “`map`” called on it (`data.map(data.map(d => d.product))`). In the destination file though, "`data`" is the name of a function called on another variable, whose results, after some chaining, have a function called on it with an anonymous function with "d" as a parameter (`svg.selectAll("rect").data(dataset).enter().append("rect").attr("y", function(d) return svgHeight - yScale(d) )`). These examples that both contain the G-AST, but where "`data`" means something different (a variable name in one, and a function name in the other) reveals a failing in our definition of G-AST containment.

  Besides these speculations, P2 questioned why one parameter was a generic name while the rest were generic nodes (4) and wondered if in d3, data can have properties (17).

- **Unfamiliar** (ids: 7, 12): For two G-ASTs, participants marked being unfamiliar with some aspect of the code. Both P1 and P2 wrote that they were unfamiliar with the graphs and nodes used in G-AST 7, and P2 wrote that they were unfamiliar with promises (a built-in JavaScript construct) in G-AST 12.

We next gave the participants the following prompt to solicit more API usage patterns: "After doing that task on all the cards, think of the ways you have used d3. What other API usage patterns can you think of that aren't here? Write down as many usage patterns as you can off the top of your head." We gave them 10 - 15 minutes to answer. P2 said they could not think of any off the top of their head, and P1 provided four patterns: 1) a generator pattern, 2) an animation pattern, 3) an interaction pattern, and 4) a switching coordinate system pattern. Our first observation on the four patterns provided by P1 was that in speaking out loud while writing down the patterns, they wrote more concrete code, and described the possible variations and purposes of different sections. So while what they wrote could be a G-AST we found, they provided out loud (and in a couple written comments), what we propose the full API usage pattern data structure would provide: a graph that represents the variations, and text to describe the purpose of sections. We then looked through each of the four patterns to see how they line up with the G-ASTs we found, and the code corpus we used. For the first pattern (generators), we searched all the code for the word “generator” and did not find that usage pattern in our corpus, or at least not with that naming convention. For the second pattern (animation), we had not generated any G-ASTs that fit, though we found examples in the corpus, so it is possible that if we ran our algorithm with more code sets, it could have found that pattern. For the third pattern (interaction), we found a G-AST that they didn’t get to (id=143) that was similar to the one they wrote down (they suggested as an example “d3.append().on(’mouseenter’, () =>)” and we found “d3.select(«...»).on(«...», function ()).” For the fourth pattern (switching coordinate system), we could find no code in the corpus that created euclidean coordinate systems like their example.

In summary, our participants described the purpose of a clear majority of the G-ASTs we showed them, but there were a number of problems with the algorithm’s output that still need to be addressed, particularly with code that is too abstract, confusing and erroneous sections of code like lines that consist only of an identifier, and potential issues with what counts as G-AST containment. As for the four API usage patterns that P2 came up with after looking at our set, one was found by our algorithm, two were not present in our code corpus, one was in our corpus but not found by our algorithm (though may have been found if it used more code sets).

4.6 API Usage Template Use Scenarios

In order to further demonstrate the usefulness of our proposed APU usage pattern data structure (and our algorithm that generates G-ASTs for that data structure), we present below 12 scenarios where we believe API usage templates will benefit users. We explain how API usage templates would be used in terms of 8 supporting data structure functions (Table 4.1). Instead of choosing just one scenario, creating a tool, and verifying its effectiveness, we believe that showing these 12 different scenarios to be a more effective demonstration of the potential use of our data structure, and which will also give more direction and inspiration to future researchers and tool designers.

1. **I want to learn an API**
   - **Description**: I want to learn what an API can do and how to do it. No specific task to start with.
   - **Current Methods**: A developer can read official documentation, read tutorials, or look through examples that they can find.
   - **Using API usage templates**: A user can view a list of all API usage templates (Function 1), which can then be organized by prevalence and similarity / containment, or they can navigate templates by jumping from one template to a related template (Function 2).
   - **Benefit of API usage templates**: This method of learning is complementary to the other methods. While it won't convey domain concepts or execution facts as well as other methods, it will be a faster way to explore the larger use patterns and we believe it will be faster than documentation or examples, and more flexible than tutorials for getting started with an API.
2. **I want to organize a curriculum about an API**
   - **Description**: A tutorial creator (e.g., teacher, API developer) wants to figure out a sequence of subjects to organize a curriculum around.
   - **Current Methods**: A tutorial creator can use own their subjective intuitions, or manually examine at how people use code in examples. This method may miss common uses that tutorial creator doesn't think of.
   - **With API usage templates**: Can generate and order API usage templates (Function 1) and organize them by prevalence and similarity / containment. Designer may deviate from this list based on their intuition or personal observations, but the generated list of templates provides an initial order, and possibly more complete set of knowledge.
   - **Benefit of API usage templates**: We believe this will save tutorial creators time and by (hopefully) increasing the templates they consider, will end up with

Table 4.1: Functions for the API usage pattern data structure in support of the 12 scenarios (section 4.6).

Function #	Description	Input	Output	Scenarios
1	List all templates of an API	none	List of templates	1, 2, 3
2	Find Related template (Related could be defined by G-AST inclusion or other AST similarity measure)	Template	List of related templates	1
3	Search all templates	search query	List of templates that with code or purpose data that matches search query	4, 12
4	List all templates that appear in a piece of code	Code sample	List of templates with mappings of template's G-AST nodes into the sample code	5, 6
5	Locate a given template in code	Code sample (if present), template	Mapping(s) of the template AST nodes to the code AST nodes (templates may appear more than once)	6, 7
6	Find potential changes to a specific program	Code sample	code change sets for adding templates that are related to the templates found in the code sample (inserting at G-AST containment matching points)	8, 9
7	Find potential changes to a specific program that match query	Code sample and search query	code change sets for adding templates that are related to the templates found in the code sample	10
8	Find near-miss templates (templates that aren't in the code, but are nearly in the code	Code sample	list of near-miss templates	11

better quality tutorials.

3. **I want to test API knowledge**
   - **Description**: A developer wants to test their knowledge of an API or a teacher wants to create a test of students' knowledge.
   - **Current Methods**: A developr can look at documentation, which provides list of API knowledge, generally execution facts. They can sort through examples manually to decide on questions. Other than that, teachers and developers must come up with their own test material based on intuition.
   - **With API usage templates**: API usage pattern knowledge can be tested by selecting API usage templates from a list of all templates (Function 1), and having test-takers either write their own description of the template code and compare to the one in the data structure (evaluating correctness of description will probably have to be done manually) or giving test takers a description of a template and having them write code to do that (which can be evaluated by looking at the descriptions of templates that show up in their code).
   - **Benefit of API usage templates**: We believe this can either automate or increase the speed and quality with which test questions about API usage patterns are written.
4. **I want to know how do I do action X in an API**
   - **Description**: A developer wants to determine how to perform a specific action in an API.
   - **Current Methods**: A developer can search sites like StackOverflow for posts related to the action (or create new post), read documentation to see if API features or official examples relate to the action, and search for example code in places like GitHub that has relevant code.
   - **Using API usage templates**: Use the API usage template data structure to search for templates that match what the user is looking for (Function 3). This

would be similar to just searching for examples online, but the information is synthesized into templates that are more abstracted.

- **Benefit of API usage templates**: We believe that our automatically generated templates would cover more situations than places like stackOverflow can, and that our templates being abstracted and synthesized examples would answer developers questions more directly than searching through examples.

5. **I want to understand a segment of code**

   - **Description**: A developer has a some example code or code in a project they are working on and they want to understand a part (or all) of the code.
   - **Current Methods**: A developer can try to read the code as is, they can look through available documentation (sometimes accessible by hovering in an IDE), and they can look for similar code in places like StackOverflow posts or GitHub projects.
   - **Using API usage templates**: We can provide annotations and explanations of code by first finding all templates present in the code (Function 4), and then displaying explanations, annotating patterns, and allowing users to select or hover over specific pieces of code and view the templates.
   - **Benefit of API usage templates**: We believe our method improves on current methods because it can automatically document, highlight, and explain sections of code.

6. **I want to know where this piece of code does action X**

   - **Description**: A developer is looking at a project and wants to isolate the part (or parts) of the code that does a certain action so they can understand or modify it.
   - **Current Methods**: A developer can read or skim the code looking for where the action is performed, use a search function in the to look for something relevant

(e.g., string search, call hierarchy navigation), or use a debugger to isolate where the relevant action is happening (e.g., WhyLine[61]).

- **Using API usage templates**: In order to search for places in the code that perform a specific action, we can find all templates in a piece of code (Function 4), and then find and display those that match the user's query. Alternately if the developer is already interested in a specific template, we can find the template wherever it appears in the code (Function 5) and show the location(s) to the developer.
- **Benefit of API usage templates**: The advantage to our methods over other methods is that with the text explanations, there is more metadata to help match the users query if they are using one, and once a match is found it is not only a match of a single location in the code, but a template that could be spread across the code and thus reveals the underlying structure that is relevant, which may be more useful for understanding and making modifications.

7. **I want to know what other code does this**
   - **Description**: A developer wants to find other code that is related to what they are currently looking at (whether other code within their project, or other code in online repositories)
   - **Current Methods**: Developers can skim code to see if anything matches or use search in their IDE or online in Google or StackOverflow. Robillard created a system that automatically finds other parts of the code that may be related to what a developer is working on [108].
   - **Using API usage templates**: Once a template is selected, look for wherever the template appears in whichever code is relevant (Function 5).
   - **Benefit of API usage templates**: While different searches have their benefits, we believe there are situations where finding the same API usage pattern would be particularly beneficial, like when looking for code to refactor, or looking for

other code as a guide on what to do.

8. **I want to explore what changes can I make to a piece of code**
    - **Description**: A developer is looking at a piece of code and wants to explore potential changes without a specific goal change in mind.
    - **Current Methods**: There are few direct ways to explore this. A developer can learn about the API and think through their own ideas of changes. Perhaps the best way to do this is by looking through examples, whether example sets in documentation, or online code repositories, or automatically discovered ones like in Robilliard's system [108], or Glassman's system for visualizing examples [37].
    - **Using API usage templates**: Our template structure allows us to find a list of potential changes to code (Function 6), which we can then display to the user as a list or as actual changes.
    - **Benefit of API usage templates**: We believe this will be the faster than other current methods for developers to consider and test many potential variations to their code.

9. **I want to know what can go in this place in the code**
    - **Description**: A developer is looking at a particular location in the code and wants to explore options for what can go in that spot.
    - **Current Methods**: Besides just learning more from documentation and looking at examples independently, there has been some research on parameter suggestion [148].
    - **Using API usage templates**: By looking at potential changes to code (Function 6), selecting those that insert something in the place the developer has selected, and additionally looking through examples that contain those templates, we can provide a list of options and examples that go in that place. Additionally we can potentially suggesting other changes that should go with that insertion if the templates involve changes elsewhere as well.

- **Benefit of API usage templates**: We believe this will be as good as other parameter suggestion methods, but with listing other concurrent changes, we expect this to help developers more with complicated changes.

10. **I want to change this piece of code to do action X**

    - **Description**: A developer has a piece of code, but they want it to perform a specific action that it doesn't yet perform.
    - **Current Methods**: The main methods of doing this are learning more about the API to piece together potential solutions, searching or posting to sites like stack-overflow, or looking for examples that might demonstrate what the developer wants. Most of these happen outside the developer environment.
    - **Using API usage templates**: We can do something similar to scenario 8, but in this case we include a query when we find potential changes (Function 7). Then the developer can look through a list of potential change sets which match the query.
    - **Benefit of API usage templates**: We believe this will be faster than other methods for developers to complete this task (as long as our data structure has the needed template with metadata to match the query).

11. **I want to know if there are any defects in my code**

    - **Description**: A developer wants to check and see if there is any code that might be a defect.
    - **Current Methods**: Other research projects look for unusual code to flag as a potential defect [107, 32].
    - **Using API usage templates**: We propose a method of detecting potential defects by find near-miss templates (Function 8), that is templates that are not in the code, but nearly match the code, we expect that popular templates that are nearly in the code are more likely to be what the developer intended and thus be defects.

- **Benefit of API usage templates**: This method is likely to produce a different set of potential defects than other methods, and thus would be valuable to use in addition to the other methods.

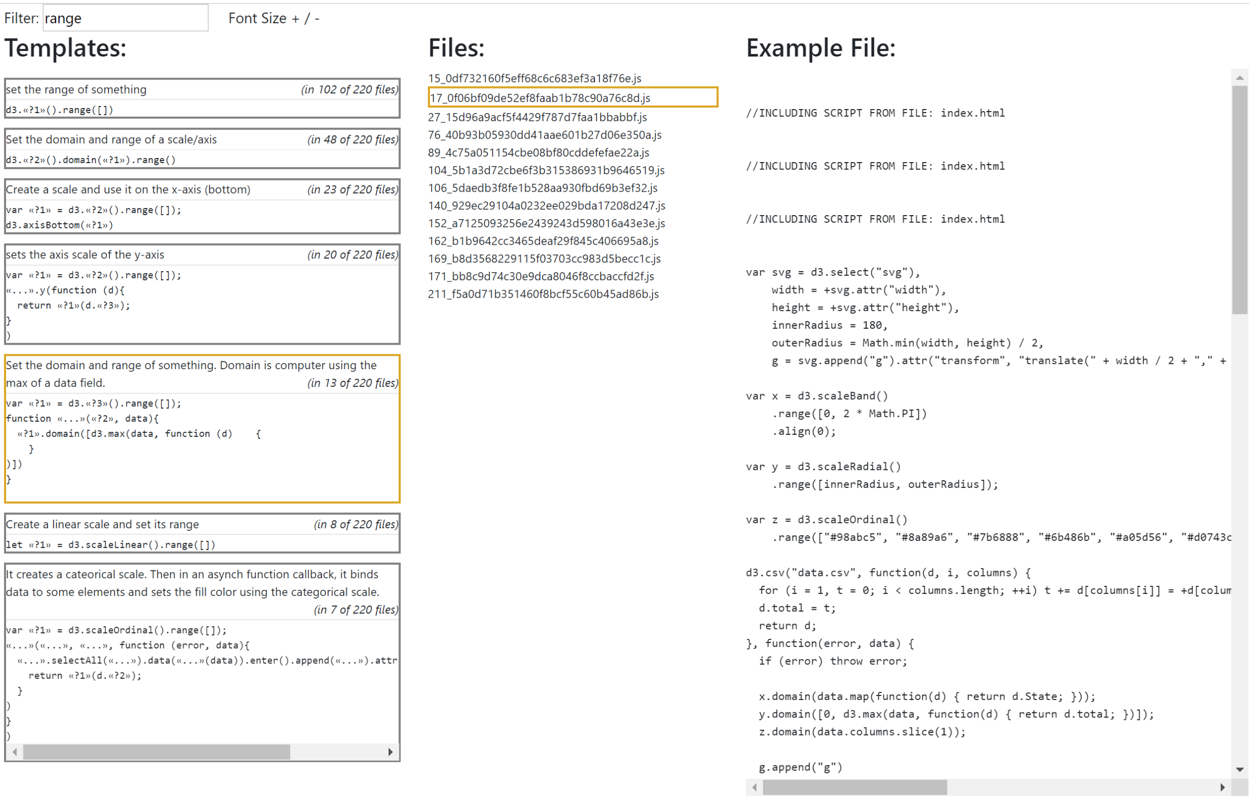
12. **I want to know which code samples are most likely to be clones**
    - **Description**: A developer wants to find similar code for tasks like refactoring, or a teacher wants to find similar code to detect cheating.
    - **Current Methods**: There are many methods of clone detection already available [10, 114].
    - **Using API usage templates**: To look for clones, we find all templates in each code sample (Function 3), then we compare which templates the code samples share in common (also accounting for the prevalence of each template and relationship between templates).
    - **Benefit of API usage templates**: This is a new method of clone detection which may be more or less effective than former methods, and may find a different viable set of potential clones.

4.7 Preliminary Template User Study

In order to test whether or not API usage templates are beneficial to developers, we ran a study that placed developers into scenario 1 in 4.6 ("I want to learn what an API can do and how to do it") and controlled their access to API usage templates. As we believe templates will aid in participants answering these questions, we believe they will find the task easier and report their answers in more detail, so we we therefore will test the following hypotheses:

- H.1: Developers will report that it is easier to answer questions about API elements when they have access to templates.

- H.2: Developers will provide more details in their answers to questions about API elements when they have access to templates.



Filter: range

Templates:

set the range of something *(in 102 of 220 files)*

```
d3.«?1»().range([])
```

Set the domain and range of a scale/axis *(in 48 of 220 files)*

```
d3.«?2»().domain(«?1»).range()
```

Create a scale and use it on the x-axis (bottom) *(in 23 of 220 files)*

```
var «?1» = d3.«?2»().range([]);
d3.axisBottom(«?1»)
```

sets the axis scale of the y-axis *(in 20 of 220 files)*

```
var «?1» = d3.«?2»().range([]);
«...».y(function (d){
  return «?1»(d.«?3»);
}
)
```

Set the domain and range of something. Domain is computer using the max of a data field. *(in 13 of 220 files)*

```
var «?1» = d3.«?3»().range([]);
function «...»(«?2», data){
  «?1».domain([d3.max(data, function (d)    {
    }
)])
}
```

Create a linear scale and set its range *(in 8 of 220 files)*

```
let «?1» = d3.scaleLinear().range([])
```

It creates a cateorical scale. Then in an asynch function callback, it binds data to some elements and sets the fill color using the categorical scale. *(in 7 of 220 files)*

```
var «?1» = d3.scaleOrdinal().range([]);
«...»(«...», «...», function (error, data){
  «...».selectAll(«...»).data(«...»(data)).enter().append(«...»).attr
    return «?1»(d.«?2»);
  }
)
}
)
```

Files:

15_0df732160f5eff68c6c683ef3a18f76e.js
17_0f06bf09de52ef8faab1b78c90a76c8d.js
27_15d96a9acf5f4429f787d7faa1bbabbf.js
76_40b93b05930dd41aae601b27d06e350a.js
89_4c75a051154cbe08bf80cddefefae22a.js
104_5b1a3d72cbe6f3b315386931b9646519.js
106_5daedb3f8fe1b528aa930fbd69b3ef32.js
140_929ec29104a0232ee029bda17208d247.js
152_a7125093256e2439243d598016a43e3e.js
162_b1b9642cc3465deaf29f845c406695a8.js
169_b8d3568229115f03703cc983d5becc1c.js
171_bb8c9d74c30e9dca8046f8ccbaccfd2f.js
211_f5a0d71b351460f8bcf55c60b45ad86b.js

Example File:

```
//INCLUDING SCRIPT FROM FILE: index.html


//INCLUDING SCRIPT FROM FILE: index.html


//INCLUDING SCRIPT FROM FILE: index.html


var svg = d3.select("svg"),
    width = +svg.attr("width"),
    height = +svg.attr("height"),
    innerRadius = 180,
    outerRadius = Math.min(width, height) / 2,
    g = svg.append("g").attr("transform", "translate(" + width / 2 + "," +

var x = d3.scaleBand()
    .range([0, 2 * Math.PI])
    .align(0);

var y = d3.scaleRadial()
    .range([innerRadius, outerRadius]);

var z = d3.scaleOrdinal()
    .range(["#98abc5", "#8a89a6", "#7b6888", "#6b486b", "#a05d56", "#d0743c

d3.csv("data.csv", function(d, i, columns) {
  for (i = 1, t = 0; i < columns.length; ++i) t += d[columns[i]] = +d[colum
  d.total = t;
  return d;
}, function(error, data) {
  if (error) throw error;

  x.domain(data.map(function(d) { return d.State; }));
  y.domain([0, d3.max(data, function(d) { return d.total; })]);
  z.domain(data.columns.slice(1));

  g.append("g")
```

Figure 4.7: The interface with usage templates and their connected examples for the preliminary study of the effectiveness of usage templates

4.7.1 Method

Interface

In order to control developers' access to API usage templates, we created a bare-bones user interface (see Fig. 4.7), which was available for three APIs: D3, ThreeJS, and OpenLayers. The interface had three panels:

- Example File (right column): This view let users view one of the example files and scroll vertically and horizontally.

- File List (middle column): a list of example files that use the API. When one of these files is selected, it appears as the Example File in the right column. These files are the set of files we used to generate G-ASTs from in 4.5. The filenames for the OpenLayers and ThreeJS files gave an indication of what the purpose of the file was, but for the files from D3, we got them from github gists, for which used the randomly generated string name of the gist for the file name (the actual file names in the gists were often unhelpful, like "index.js", and we combined all the js files into one larger file.

- Templates (optionally hidden left column): A set of API usage templates. When one of these templates is selected, only files that include that template are visible in the File List (middle column). The code templates themselves were the random set of automatically generated G-ASTs from our evaluation of G-ASTs in 4.5 (8 for our demo with OpenLayers, 28 for ThreeJS, and 29 for D3). We added the expert description of code purpose from D3 from that study, and for OpenLayers and ThreeJS, descriptions of code purpose were written by the main researcher. We also added information about how many of the example files the template is used in.

Additionally we added a filter box filter box that filters templates and the file list based on whether the templates or files contain the text the user enters in the filter box. Also, since we made the interface with small text, we also added a font size +/- button so users could increase or decrease the size depending on their needs.

Procedure

In order to give participants a task matching scenario 1, we gave participants questions to answer about specific API elements.

- **"What does [API Element] do?"** This is one of the most common questions asked about an API, which we derived from Q.4 in Duala-Ekoko and Robillard ("What is the functionality of a given API type?")[31].

- **"Which other [API] elements are used together with [API Element]? How are these other elements related to [API Element]?"** We derived these questions

> from Robillard and DeLine finding the importance of understanding "ways to use a number of related API functions together"[110], and Q.7 and Q.13 in Duala-Ekoko and Robillard ("How is the type X related to the type Y?" and "Which other API elements are necessary to use a given API type?")[31].

For the study itself, after obtaining informed consent, we demonstrated the interface on one API (OpenLayers), and then we had each participant answer questions about two API elements each on two APIs: d3.js (`select` and `range`) and ThreeJS (`Three.Mesh`, `Three.Scene`). For one of those two APIs participants got the column of API usage templates, and with the other API that column was hidden (though we kept the other two columns at only 1/3 of the screen width to not give extra screen size benefit to not having the templates column). We randomly balance the conditions of which API is asked about first and which API has the templates.

When answering these questions we limited our participants to only gathering information from our interface and not using outside resources. Since we believed the API usage templates to be an improvement on examples and since they were patterns derived from the examples, we wanted it directly compare using our templates with the examples to just directly using the examples themselves (note that just adding information doesn't necessarily help and can slow people down as found in API knowledge study in 3.4).

After each set of answering questions about an API, we ask our participants to answer follow-up questionnaire in a semi-structured interview where we ask them "How difficult was it to answer the questions?" on a 7-point Likert scale, then we ask them open-ended questions about what strategies they used in answering the questions, how they would improve the interface, and what additional information would be helpful in answering the question.

Participants

We recruited eight university students who were at least 18 years old and knew some JavaScript but hadn't used either d3.js or ThreeJS. Our participants ages ranged from 19 to

23 ($\mu = 25$), 5 identified as male and 3 as female, and their year in school ranged from 2nd year of undergrad to 2nd year PhD student.

Analysis

Task difficulty In order to measure the effects of providing templates on developers' perception of task difficulty (H.1), we created a regression model. The unit of analysis was a participant / API pair (eight participants and two APIs give us 16 participant / API pairs). We used perceived difficulty as the dependent variable and for independent variables we used presence of templates, which API.[8] Since our dependent variable (perceived difficulty) was an ordinal Likert scale, we used proportional odds logistic regression models. We also used participant ID as a random effect. We then used analysis of variance (anova) to compute statistical significance.

Claims about API elements In order to measure the effects of providing templates on developers' answers to our task questions (H.2) we needed a way of categorizing their answers. To do this, the author of this dissertation went through each of the 4 API / element pairs (D3 - select, D3 - range, ThreeJS - THREE.mesh, ThreeJS - THREE.scene), and then went through all the participant answers for the API / element pair, and made a code book with a code for each separate claim made about that API / element pair in the answers. In order validate the code book, a second researcher coded a random selection of half the answers and the two researchers resolved inconsistencies in their codes and changed the code book as needed. Then both researchers coded the other half of the answers independently. Our inter-rater reliability check showed that 80% of our codes matched. The two researchers then discussed and resolved all differences in coding.

After we completed the coding, we counted the number of claims each participant made about each API element as a crude measure of how much progress they made in answering

[8] We chose not to use other potentially related measures we had for independent variables like previously learned API since our dataset was so small.

the questions. We then created a regression model for number of claims. This model was similar to the previous one for perception of task difficulty, except the independent variable was number of claims, the unit of of analysis was a participant and API element pair (eight participants and four API elements gave us 32 participant / API elements), and it was a generalized linear model since the distribution of number of claims was not significantly different from a normal distribution (Shapiro-Wilk normality test, $w = 0.97$, $p < 0.45$).

Next, we examined which specific claims occurred more often in those with templates and those without templates to see if any patterns emerged. Since in each case there were four participants with templates and four without templates, we focused on claims that were made by at least three more people in one group than in the other, as well as sets of related claims that skewed in one direction.

Using the interface We asked open-ended questions about our participants experience with using the interface to answer our task questions twice each, for a total of 16 interviews. The average length of the interviews was 5.3 minutes (total 84 minutes).

To understand how our participants made use of our user interface, particularly the templates, we analyzed transcriptions of our recordings of our participants' answers to questions about their strategies and thoughts on the user interface. The author of this dissertation coded the responses as to what part of the interface they addressed (Templates, File List, Example File, Search / Filter[9], or Other), and whether the participant was referring to 1) something they used or that worked, 2) something that didn't work or was a problem, or 3) a suggestion for a change to the interface or content.[10] In reporting participants' responses, we combined things that participants said didn't work with suggested changes, because they seemed to us to be addressing the same concerns, just with a different framing. The

[9] We group together both our provided search box, and the use of the browser's find feature (ctrl-f or cmd-f) into one Search / Filter category.

[10] This categorization process was straightforward since we could clearly identify in these short interviews which part of the interface participants were discussing, and they were specifically discussing their experience in terms of what worked well or didn't work well or what changes they would want (which was how our questions were framed).

researcher then summarized some of the comments made by participants in each of these categories to point toward understanding how the interface (and particularly the templates) worked well, or failed participants, and how we would improve the interface in follow-up studies. Finally, the researcher summarized some of the comments that fell outside of our pre-set categories.

4.7.2 Results

Task difficulty

Table 4.2 shows the resulting model from our regression on perceived difficulty in relation to whether a developer had templates and which API they were using. We found that participants who had templates rated their task as significantly easier than those who did not have templates ($\chi^2(1, N = 16) = 4.6$, $p < .03$). We also found that participants rated ThreeJS as significantly easier than D3 ($\chi^2(1, N = 16) = 6.4$, $p < .01$).

When looking at the data itself (see Figure 4.8) we noticed that for ThreeJS, the four who had templates rated it as a difficulty level of 2, and the four who did not have templates rated it as a difficulty level of 3 or 4. On the other hand, for D3, the participants' difficulty rating spanned a larger range (2 to 6) without any clear connection with having templates.

Claims about API elements

Table 4.3 shows the resulting model from our regression on number of claims made in relation to whether a developer had templates and which API element they were answering questions about. We found no statistically significant difference in the number of claims depending on who had templates ($F(1, 20) = 2.4$, $MSE = 7.0$, $p < .13$). On the other hand, which API element was being used did have a significant effect ($F(3, 20) = 4.3$, $MSE = 12.4$, $p < .02$), with our model showing D3-range having the smallest number of claims.

We next looked at which claims were particularly common in either the four who had templates or the four that did not for each API. The majority of claims were only made by

	model		anova		
	β	z	df	χ^2	p
Has Templates	-2.5	-1.6	1	4.6	$<.03$*
API (ThreeJS)	-3.2	-1.6	1	6.4	$<.01$*

Table 4.2: Ordinal mixed regression model and anova for perceived difficulty in each API, testing the role of having access to templates. Note: API is compared against D3.

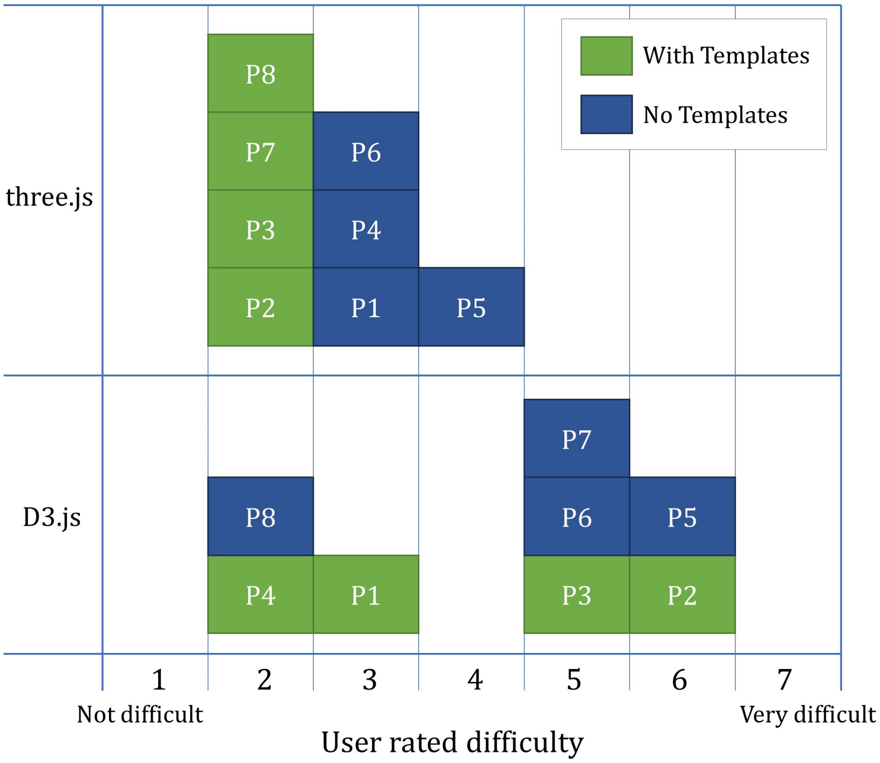


Figure 4.8: Perceived difficulty of answering the task questions about ThreeJS and D3. Each box represents a participants' response, with colors indicating whether a participant had access to templates or not.

		model		anova		
		β	t	df	F	p
	Has Templates	-0.94	-1.6	1	2.4	<.13
	ThreeJS-scene	-0.2.5	-0.30			
API element	D3-select	-0.88	-1.0	3	4.3	<.02*
	D3-range	-2.8	-3.2			

Table 4.3: Linear mixed regression model and anova for number of claims made about each API element, testing the role of having access to templates. Note: API elements are compared against ThreeJS-Mesh.

one participant (77 / 135), and there were only a few instances where claims happened at least 3 more times in one group, or groups of claims that skewed in one direction. Here we list the ones we noticed, along with our observations on how our templates relate to those claims:

- D3-select *works with DOM / HTML elements* (none of the four with templates mentioned this, all four without templates mentioned this). None of our templates mentioned the DOM by name and only one unrelated template mentioned HTML by name. In this case, we think it is likely that all participants knew this library worked with the DOM / HTML, but those who had templates followed the descriptions of the templates, which didn't explicitly mention this.

- ThreeJS-mesh (2 related claims) *can be added to a scene, and doing so makes it a part or layer of a larger object (that is, the scene)* (both mentioned by all four with templates, and only one without templates). Several of our templates demonstrated adding a mesh to a scene, so we believe this is why they mentioned it.

- ThreeJS-mesh (8 claim group) post-creation actions. All claims about what can be done with a Mesh once it's created (including adding it to a scene above) are mentioned by those who have templates as much or more often than those who did not have templates (6 more often). Those without templates appear to have focused more on what goes into creating meshes (see the next bullet point). Our interface had multiple examples of code using Meshes after they are created and stored into variables.

- ThreeJS-mesh (9 claim group) using Materials. All claims about using a material were used at least as often by those without templates (7 more often). While we had some templates that showed details about creating a material, and we had templates saying that a Mesh's constructor took a material, we had no templates that showed both details of creating a material and then using it to create a mesh.

- ThreeJS-scene (2 related claims) is used with the render() function, and rendering creates a visual display of the objects in the scene (both mentioned by three without templates and never mentioned by those with templates). While some of our templates showed the render function, none of those templates included the scene.

Using the interface

Below we will summarize and quote some of the comments made by our participants about how each section of our interface was either useful and used, or didn't work and should be changed. We will start with their comments on example files, the file list, and the search / filter functionality before we talk about templates, seeing what our participants got out of those other interface elements first. This way, we can better understand how templates, when they were available, aided with, replaced, or were irrelevant to participants' efforts with the rest of the interface. After seeing what they said about the rest of the interface, we will then summarize our participants' comments on templates before finishing with a summary of some other comments that didn't fit into our categories. As we quote participants we will give their participant number, which API they were answering questions about and whether or not they had templates. We also make minor edits to quotes for clarity.

Example File Our participants described a number of ways they found the example files useful in answering our questions about specific API elements. For example, the example files could provide " concrete examples" of using the API element (P4, D3, with templates), and a "context in which [the API element] was used" (p7, D3, no templates). Additionally the example code sometimes included helpful comments (p2, D3, no templates), and the example could be based on familiar examples, like p4 found,"you can kind of use those context clues [specific settings of moon, earth radius, etc.] to figure out what a mesh is, what geometry is" (p4, ThreeJS, no templates).

Some participants mentioned using multiple example files to answer the questions, like P6, who said, "I would try to make sure to look at maybe like two or three different types of files and see where these functions were being used in those different files " (P6, ThreeJS, no templates), and P8, who said, "I think finding many examples told me the potential ways I could use [the API element" (P8, D3, no templates). Some specifically mentioned finding patterns by looking at multiple files, like P3 who described "writing down examples of where I found [the API element being used" to build a mental "schema" of how it is used (P3, D3, no templates), P2 who "saw the same sort of pattern across multiple [files]" (P2, D3, no templates), and P1, who said: "After clicking through a couple, you could see what trends occurred" (P1, D3, with Templates).

Participants found some limitations with using the example files or had suggestions for making them more useful. One said that "sometimes the files were a little confusing," and that they "would get a little sidetracked by things [they] didn't understand," (P2, ThreeJS, with templates). One said that "searching through the file on this one without having any other context actually did make it a lot harder to understand" (P3, D3, no templates). Two specifically said they wanted more code comments (P4, ThreeJS, no templates; and P6, ThreeJS, no templates), and one said they wanted information on "what the code is used for" (P5, ThreeJS, no templates). Using multiple files didn't always help, one participant said, "it seems like [these functions] were being used really similarly throughout the different files, at least the ones I was looking at, and so I didn't get too much more information from multiple files than I would have gotten from just one" (P6, ThreeJS, no templates).

File List There weren't many comments specifically on how participants the file list portion of the interface and found it useful (though participants previous mentions of looking at example files implies that they used it). One participant mentioned selecting files from the list "randomly" (P3, D3, no templates; and p2, ThreeJS, with templates). Another mentioned selecting files based off of an incorrect assumption about how the files were sorted (they were sorted alphabetically by the name of the file, not intended to have any deeper meaning): "I looked through several of the top files, which I assume are like examples of using the library, whereas some of the ones deeper down here are more like source code or lower-level implementations" (P4, ThreeJS, no templates). Finally one participant said that with the ThreeJS library, they found the file names helpful: "it was nice to have file names, because then I could kind of get an idea of what the general file was trying to do" (P6, ThreeJS, no templates).

There were a number of complaints or suggested changes to the file list. One participant complained that "in general, like a very long file list is not great" (P3, ThreeJS, with templates). One participant complained about the random order and suggested that the file list could use "maybe some sort of sorting of the files [...] so maybe having the most relevant files or the simplest files or something other people found helpful toward the top might be a little easier " (p2, ThreeJS, with templates). Another complaint was that the file list was too large on the screen, so "it's taking up a lot of space that I would much rather be looking at the examples" (P6, D3, with templates). One participant said that when a template was selected, rather than hiding the files that didn't match, they wanted to just lighten the non-matches to de-emphasize them (P7, ThreeJS, with templates). Finally there were a number of participants who singled out the file names for D3 as being unhelpful[11]

[11] These files came from github gists, where the gist only had a random string as the project name, and the JavaScript files were in the gist were often just index.js. For each gist, we combined all js files into one example file and gave it the randomly generated gist name as the file name with a number prefix added to aid in recognizing the filenames. See Figure 4.7.

(P3, D3, no templates; P5, D3, with templates; P6, D3, with templates; and P8, D3, no templates).

Search / Filter Participants made many mention of using the search and filter functionality, generally with the name of the API element we asked about, like "I typed that word [the API element name] into the filter" (P2, D3, no templates), and "I would randomly select the example files and use control + F to see where they were used in those files" (P6, D3, with templates). P7 explained that the filter was helpful because, "it was nice to see which file had [the API element], I guess. So I didn't have to click on just any random file hoping that it used 'range"' (p7, D3, No templates). One participant mentioned wanting to learn more about a related API element and using the filter to learn more about that (P6, D3, with templates).

There were a number of different aspects of search and filter that participants said were problems or that they wanted to change. For example, some, like P7, said they didn't like that filter and search were two separate steps: "If I'm filtering for 'scene.add,'then I shouldn't have to also type in control-F 'scene.add' on the file" (P7, ThreeJS, with templates). There were also complaints about how the browsers' search looked at the whole page and not just the example file (p2, ThreeJS, with templates), or how it found unrelated things, like a search for "range" found "orange" (P2, D3, no templates). Some of the suggestions for improving search and filter included making a separate search for just the example files (P4, ThreeJS, no templates), or making the filter automatically highlight (P1, D3, templates), snap to (P2, D3, no templates) or show examples of (P3, D3, no templates) matches to the filter.

Templates There were a number of ways participants mentioned using templates and finding them useful. For example, a couple said they had used templates to filter which files to look at (P5, d3, with templates; and P7, D3, no templates[12]), with P7 starting with the

[12] P7 was comparing their latest task without templates with how they had used templates to filter in their first API.

most common templates and moving on to the less common templates[13]. Some participants talked about they type of information templates gave them, whether it was “context” (P3, ThreeJS, with templates; and P4, D3, with templates), “information about how this entire library is supposed to work” (P2, ThreeJS, with templates), “[explanations] of what you can do with functions” (P1, ThreeJS, no templates[14]), and “the only thing that kind of gave [an] English word description of anything.” Additionally, participants said templates provided a “skeleton” (P6, D3, with templates) or “generic version” (P3, D3, no templates[15]) of function use, with P3 further saying without templates they would “have to go through like a bunch of different files to find it and piece it together on [their] own.” Some participants said that after reading through templates, they would move onto to the files. For example, one said they would then “[dive] into the code examples]” to test their “mental model” from the templates (P2, ThreeJS, with templates), and another said, “I would first look at the templates, and then if I was confused about specific variable usage, I could then look into the example files” (P8, ThreeJS, with templates). Finally, some participants, when they did not have templates, expressed their desire to have access to templates again (P2, D3, no templates; P3, D3, not templates; and P5, ThreeJS, no templates).

Our participants also mentioned many shortcomings or potential improvements to our templates. Some mentioned limitations on the usefulness of templates. For example, P1 said, “I basically ignored the whole templates thing and went straight to the files [...] I didn’t really read them that much, but I like read the [description]” (P1, D3, with templates), though when they had no templates on the next API they said, “Now that I’m thinking back on it, maybe the templates were pretty helpful because it like explained what you can do with functions” (P1, ThreeJS, no templates). Others said templates gave “a very incomplete picture of what a library is” (P7, ThreeJS, with templates), or left terms like “geometry” undefined (P3,

[13] Templates were sorted in order of how commonly they appeared in example files

[14] P2 was comparing their experience without templates with what they felt they had gained from templates previously, even though previously they hadn’t felt they used them much.

[15] P3 was comparing the templates they saw in the demonstration with what they wish they had in their task with no templates.

ThreeJS, with templates), or that while “templates were a very high-level overview, [until they] started looking at the code in the files, it didn’t really make a lot of sense what they were doing” (P2, ThreeJS, with templates). Some wanted to improve templates with more or better descriptions (p4, D3, with templates; and P6, D3, with templates)[16], and one wanted a clearer understanding of our special notation for generic G-AST nodes (P2, ThreeJs, with templates). Some participants mentioned visual design choices we made, with one saying the template descriptions didn’t stand out visually and “kind of meshes with the code, and [they] feel like that’s one of the reasons why [they] didn’t want to use the templates (P1, D3, with templates), and another saying they didn’t like that “visually, it was all black and white” (P2, ThreeJS, with templates). Finally a couple participants wanted us to connect templates better with code and other templates, one saying, “maybe [the interface] would be able to tell me like this is the segment of code that we think is a match for the template” (P2, ThreeJS, with templates), another saying they wanted a way to see what could go in the `<...>` generic nodes, and connect templates that seem to be variations of each other (P6, ThreeJS, with templates).

Other comments Besides the statements that fell into our categories above, there were a few more comments that we believe help us in understanding our results and improving our interface. For example, there were a couple comments about not being clear on the questions, like, “the only difficult part was knowing how in depth I should be going” (P7, ThreeJS, with templates). There were also a number of comments about participants using their prior knowledge, whether of English terms, like P2 saying, “scene is also an English word that people use, so I could kind of get a picture of that” (P2, ThreeJS, with templates), or of domain knowledge, like P3 saying, “just from context from working on some graphics stuff, I know that a mesh is generally a shape” (P3, ThreeJS, with templates). Some also mentioned failing to connect to prior knowledge, like P7 saying: “I was trying to equate

[16] P4 was particularly confused by a description where it seems they misunderstood our use of the word “encounters,” where we had intended it as a verb but they read it as a plural noun.

[range] to a more familiar concept, but there wasn't one concept that I could really equate it to" (P7, D3, no templates), or P8 saying, "I didn't know some of the jargon specific to this library" (P8, ThreeJS, with templates).

There were also some requests for changes to our interface that didn't neatly fit into our predefined categories. Some suggested adding visual output (P4, D3, with templates; P6, ThreeJS, no templates; and P7, ThreeJS, with templates) or adding an interactive editor (P4, D3, with templates; P8, D3, no templates; and P8, ThreeJS, with templates). One suggested adding "a high-level description of what the library is" (P2, D3, no templates), and another suggested a "dictionay" (P3, ThreeJS, with templates:). Finally, several suggested explanations of API elements (P7, D3, no templates; and P2, ThreeJs, with templates), and hover-over tooltips to show that extra information (p4, ThreeJS, no templates; and P8, D3, no templates).

4.7.3 Study Discussion

Though this was a preliminary user study with only eight participants, we still found some evidence pointing to the usefulness of API usage templates, at least in regards to the scenario we tested (scenario 1 in 4.6: "I want to learn what an API can do and how to do it"). We found that having access to API usage templates significantly decreased our participants' reported difficulty on answering our questions (confirming H.1). We did not find a significant relation between having access to templates and to how many claims about API elements participants made in answering our questions (not confirming H.2), but we did notice some specific claims and groups of claims that were found more often with or without templates, which means templates might direct participants towards or away from specific information. We also saw from the user comments, how different participants talked about how they used templates, what information they got from them, and how some of them wanted templates back when they didn't have access to them.

Looking more specifically at what participants said about our templates benefits and failings, we can see that at their best, they provided a high level overview of using the API, with generic versions of API calls, and that the English descriptions helped make sense of the purpose of the code. Our templates in effect replicated what some participants said they tried doing with the other parts of the interface, particularly when they didn't have templates: randomly select files from the list and piece together the general pattern and the purpose of how the given API element was used in the files. Thus our templates saved participants this effort of reconstructing this knowledge from the example files. Our templates also helped participants select files to learn from (an alternate to just filtering for files that mention the API element).

At their worst though, our templates only gave a partial picture of the API, the descriptions were too short, lacked detail, or were confusing, and the templates were not well connected to concrete uses of the API, the example code, and to each other. Participants soon ran out of useful information they could glean from the templates and were then forced to turn to the example files to try to fill in the gaps left by the templates themselves (whether from generic nodes, or insufficient descriptions of purpose).

Looking at the results of this preliminary study of the role of API usage templates in answering questions about an API, we believe there are some clear ways to improve our interface.

- We could increase the number of templates to cover more scenarios. For example, our small random selection of ThreeJS templates provided no template that showed scene and render being used together.

- We could improve the descriptions of what each template does by making them more in-depth, or perhaps linking to terminology and a dictionary, or facts about specific API elements.

- We could make the descriptions and the code more visually distinct from each other to make it easier for users to navigate.

- We could allow users to see what possibilities go into each generic node in a template.

- When a user selects a template and an example file with that template, we automatically locate and highlight where the template appears in the file.

- We could link related templates to each other and help participants navigate between these related templates.

Besides improvements to templates, there are also a few other improvements to the interface we could make based on our findings:

- Reduce the screen space used by the file list, so that the example files can use more screen space.

- Remove our dependence on the browser's search (e.g., control-F) by improving our filter box.

- Add an interactive editor or at least show example visual output from the example files and the templates.

Overall, we feel that this preliminary study has given initial indications that API usage templates, and our particular interface using them, can be beneficial in answering questions about APIs. We also now have feedback with points us toward a number of improvements that could be made to our interface and our templates for testing with future studies. We believe the next steps for this interface would be to implement these improvements and continue testing scenario 1 as well as testing the other scenarios we outlined.

4.8 Discussion

In this chapter, we introduced API usage templates, a novel data structure for representing API usage patterns. We then presented an algorithm for automatically extracting part of that data structure (generalized abstract syntax trees: G-ASTs) from code examples and checked their validity with expert API users. We finally presented 12 scenarios where we believe this data structure would be beneficial and found preliminary evidence that API usage templates are beneficial in at least one of those scenarios.

4.8.1 Benefits of the API Usage Template Data Structure

We believe our novel definition of an API usage template data structure represents an improvement on prior work by more flexibly representing code structures. Prior work used predefined [37] or simplified code structures [107]), while our G-AST based structure allow for more complicated code structures. Additionally, we gave space for code explanations, which are important in making use of these templates [31]. We also believe our data structures can support even more use cases than previous data structures. Our outline of 12 scenarios show a clear benefit to having the data structure in many different tasks. It is possible in some of the scenarios we listed, using our data structure may be less feasible or less useful than we expect, but we believe this data structure (or whatever supersedes it in the future) opens new doors to what is possible. There are many ways this data structure could be created (algorithmically , crowd sourced, etc.), and we present an algorithm which starts extracting part of the structure (G-ASTs) from example code. In some of the scenarios we outlined, our suggested uses of the data structure may possible from just our limited ability to create G-ASTs so far (e.g., scenario 8: I want to explore what changes can I make to a piece of code, scenario 12: I want to know which code samples are most likely to be clones), while others (like scenario 5: I want to understand a segment of code), may have limited utility without text descriptions. When we did add text descriptions to our G-ASTs to make, we found preliminary evidence that participants found tasks in scenario 1 easier with API usage templates.

4.8.2 Feasibility of Creating API Usage Template Data Structure

Though we set out an ambitious design of a novel data structure to represent API usage patterns, in this work we only attempted to extract part of the information needed to create this data structure: the G-ASTs. We must therefore judge both our efforts at generating

G-ASTs, as well as consider additional work that would be needed to fully create an API usage template data structure.

For the G-ASTs, we believe we have shown very promising early results. When we created API usage templates out of our G-ASTs generated from random selections of example files, we found preliminary evidence that developers found it easier to answer questions about an API (though they did not overall provide more details in their answers). Additionally, a significant majority of our G-ASTs were clear enough for our experts to describe the purpose of, and many of their complaints about problems with the G-ASTs were due to generic function definitions and function calls, both of which could be addressed by improving our trimming and filtering algorithm. Removing single lines that have only a single variable name would address another set of complaints fairly easily, and looking back over the templates, we believe the templates judged as “incomplete,” are still valuable in representing a coherent piece of code that may be composed with other pieces, even if it is not valuable on its own. Of course, tailoring an algorithm to work for a specific API and set of templates has the risk of not generalizing. Looking through the templates generated with the other two APIs we see some of the same problems appearing, so we believe the changes would improve it, at would likely then work for least for some other JavaScript APIs as well.

There are a number of additional changes that we believe would improve the results of our algorithm. For example, when looking at some of the G-ASTs participants marked as incorrect, we noticed that our algorithm for G-AST containment mapped to unrelated pieces of code that happened to share the same name even though they were in unrelated program scopes. A stronger test of variable relatedness may be necessary for avoiding these kinds of false-positive matches in code. Another improvement, in JavaScript at least, would be to handle the way imports allow API features to be refereed to by different names (our parser just failed on imports, so we ignored code with them). We also know of API usage patterns in some APIs that involve multiple files and filenames, and at times complicated naming conventions (e.g., Angular and React in JavaScript, Ruby on Rails). Our algorithm could be improved by making filenames and folders part of the G-ASTs and also by allowing for partial string matches. Finally, the library we are using to generate ASTs can also generate ASTs for other programming languages, but we have not tested any code for those yet and each language would probably require its own customization.

While our algorithm shows promise in generating meaningful G-ASTs, our algorithm still depends on having a corpus of example programs to find all the G-ASTs that a developer or learner would need. We chose d3 to test, in part because we knew there example code was relatively easy to find online, and we only used one of many ways of finding code for d3 (gists with a specific d3 version inclusion string). In this corpus, we did not find all of the API usage patterns mentioned by our participants, and our set of 200 file comparisons did not produce templates for all the patterns that were present in the corpus. Developing a complete list of API Usage patterns would involve finding larger corpuses, which might not be possible for some APIs (especially if the code needs to have proper licensing for reuse), and would require much more than just 200 file comparisons.

Beside the G-ASTs we generated (with some success), we need several additional pieces of information to fully automate making the data structure, such as the graph of the relation between G-ASTs, and text descriptions of the purpose of the G-ASTs, and style and whitespace information. The relation between graphn G-ASTs we believe would be easy to generate, just by checking all G-ASTs to find which ones are contained in others, and storing those relationships as a graph (though again, we may need to refine our definition of G-AST containment). Additionally, this data structure could include all the source examples (likely leaf edges), which could provide additional relevant information to people using our data structure. The second, generating text descriptions of the purpose of the G-ASTs, we expect to be challenging. This may be possible by depending on work that has been done to automatically summarize code [27, 40, 39, 76, 77, 79, 78, 84, 90, 123, 124, 145, 72, 5, 28], and by leveraging comments within the code and any text associated with code to find key words and phrases. Still, we have not tried this task yet and don't have a sense of how feasible it would actually be.

4.8.3 How API Usage Templates Could Change the Way We Use APIs

Creating these API usage template data structures (and our partial progress in creating them) could open up new possibilities in how developers learn APIs and interact with them in programming environments. Researchers have already cast visions of, and worked toward automatically generating tutorials and documentation for APIs [112, 129, 130], and we believe our API usage templates could be an important component of that. Our data structure represents one of the types of knowledge programmers need to know (see Chapter 3), and provides the ability to organize and explore this information. While programming environments have advanced from simple text entry to IDEs, like Eclipse that support auto-completion of strings with connected documentation, our new data structure could support not just auto-completion and documentation of strings but of entire templates. While prior work has done some work with allowing developers to interact with templates, these had to be created manually [89], or were restricted to simple code structures [86, 113]. Our flexible (and partially automatically generated) data structure could support similar tasks at both larger and more fine grained scales, changing the way users interact with API code. What remains to be done is further improvements in automatically generating API usage templates, the creation of tools to support different developer tasks (like outlined in our 12 scenarios), and finally testing whether developers benefit from these tools.

Chapter 5

DISCUSSION & CONCLUSION

As the demand for software developers continues to grow, more and more people will try to enter the software industry, and alternate entry routes, such as coding bootcamps and MOOCs, are likely grow (see Figure 2.1). People taking these different entry routes will face many barriers, some specific to their route and some general. In this dissertation we chose to focus on the practical knowledge barriers people face. As more and more people hit these practical knowledge barriers, researchers can help find ways to help people get past those barriers, and do so at the scale required for this large number of people. In this dissertation, I, along with my co-authors, contribute to this effort by identifying knowledge barriers (Chapter 2), deepening our understanding of what the knowledge is (chapter 3), and represent that knowledge in a data structure that we have started to be build at scale, and use that data structure to make one knowledge barrier easier to cross (Chapter 4).

We can only attempt to address barriers that we know exist, and since coding bootcamps had not previously been studied for peer-reviewed publication, we did not know how the barriers there would compare to other previously studied areas. So in chapter 2 we identified the practical knowledge barriers (and other barriers) faced by coding bootcamp students. We identified the API knowledge barrier as particularly important for bootcamp students (we discuss more in chapter 3, and, of course, API knowledge is a well known barrier outside of bootcamps). We also identified other knowledge barriers faced by coding bootcamp students, like how to network with others in order to find jobs, how to fit in culturally in the software industry (e.g., stereotypes, jargon), and how to work on programming teams using collaboration tools like git. Of course, we only studied students at a small number of bootcamps at a particular time, and more work needs to be done to understand barriers faced by bootcamp students (such as on-line vs. in person bootcamps). Each of these barriers needs to be more fully understood in order to be addressed, and addressed at scale, and in chapter 3, we choose one to focus on.

Of the many barriers we identified as being relevant to coding bootcamp students, we chose the barrier of API knowledge to explore in chapter 3. We deepened our understanding of API knowledge by building a theory of robust API knowledge comprised of three components: Domain Concepts, Execution Facts, and API Usage Patterns. This theory helps us not only organize prior findings about API knowledge (see 3.3.3), separating that knowledge into components with clearly defined relationships with each other, but also helps us understand how that knowledge is used in working with APIs (see Table 3.1). We also used two studies to evaluate hypotheses derived from our theory and found mixed evidence to support it. More work is still needed to evaluate our theory, testing if hypotheses derived from our theory are true, and in determining if our theory is useful in generating interesting questions or valuable insights. Additionally, our theory may need further refinement, in particular in regards to the evidence found in our studies, such as that the usefulness of pieces of API knowledge may be very context dependent.

With the clearer understanding of API knowledge our new theory provides, we can then consider each component of API knowledge and consider what means we already have for conveying that knowledge and how we could build scalable solutions (such as on-demand documentation [111, 129, 130]) to lower barriers to that knowledge. Some scalable solutions have already been created, such as tools that have begun to automatically extract code patterns and their alternatives (e.g., [36]), and program analyses that can automatically extract some execution facts about API behavior (e.g., [51], [148]), and our new work on API usage templates in chapter 4. Still, more work is needed for lowering the barriers to learning each component of API knowledge and their interrelations, and we believe our theory will help guide the creation and testing of these new tools. Additionally, our theory specifically excluded a number of components of API knowledge that can also act as barriers, and would need their own scalable solutions such as: knowing how to install the API and execute code, knowing how to write code for others to read and what community standards exist, and knowing how to take part in the API community.

Our work in chapter 4 demonstrates a way we can use the theoretical understanding of API knowledge we developed in chapter 3 to start building a scalable solution to one of the components of API knowledge: API usage patterns. We used our theory to define a new data structure (API usage templates) to represent API usage patterns, which specifically includes places for storing the rationale for the API usage patterns (based on our theory), which is something we didn't find in prior work on usage patterns. Additionally, while prior work used predefined [37] or simplified code structures [107]), we created Generalized Abstract Syntax Trees (G-ASTs), which are more flexible and allow for more complicated code structures not supported by this prior work (such as in Fig. 4.1, which has a constructor in a constructor in a constructor). We believe this data structure, however it is created, aligns more fully with our theory of API knowledge, and therefore will be of greater benefit to developers (as outlined in 12 usage scenarios in 4.6).

In order to create our API usage template data structure in a scalable way to meet the demands of many developers using many APIs, we began work to automatically extract part of that structure (G-ASTs) from example code. While our algorithm needs more refinement, the process itself could easily scale as long as example corpuses can be found for APIs. When we manually added code summaries, we found preliminary evidence that developers did find answering questions about APIs easier (one of our 12 usage scenarios). More work is still needed to see if this data structure helps in other scenarios, and in designing the UIs implied by our usage scenarios, and in implementing the 8 supporting functions for those scenarios (Table 4.1). More work is also needed to create the code summaries in a scalable fashion (e.g., automated generated along the lines of prior work on code summaries [27, 40, 39, 76, 77, 79, 78, 84, 90, 123, 124, 145, 72, 5, 28] or crowd sourced). Still, this is the start of work that would be needed to lower this barrier for large numbers of people entering the software industry and for the large number of APIs they need to work with.

Through this dissertation, we demonstrate a template for creating scalable solutions to addressing practical knowledge barriers in professional programming. We do this by identifying barriers, understanding through theory building, and finally using the new theories to create scalable solutions to these barriers. We believe it was particularly important that when we sought to identify knowledge barriers in chapter 2, we focused on coding bootcamps as opposed to traditional college degrees in computer science. We believe studying coding bootcamps is essential because bootcamps are one of the alternate routes into the software industry that have opened up due to the high demand for software developers. In particular, bootcamps have a different style of education and attract a different (and more diverse) population into the software industry. If we want to lower barriers to professional programming, it is important that we understand how these barriers are faced by different groups using different routes, and while there have been numerous studies on traditional computer science college degrees, ours was the first peer-reviewed publication on coding bootcamps. As we begin to address knowledge barriers at scale, we need to make sure we are targeting the needs of diverse populations, and we will also need to make sure that our solutions are benefiting all and increasing equity, and not disproportionately benefiting the already privileged [41]. If we do this, we can lower barriers for all, so that the income and career opportunities provided by the growing software industry are truly open to all.

BIBLIOGRAPHY

[1] Stack Overflow developer survey 2016 results, 2016. URL: `http://stackoverflow.com/research/developer-survey-2016`.

[2] npm, 2018. URL: `https://www.npmjs.com/`.

[3] Stack Overflow Developer Survey 2018, 2018. URL: `https://stackoverflow.com/insights/survey/2018/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share`.

[4] OpenLayers - Welcome, 2019. URL: `https://openlayers.org/`.

[5] Fouad Nasser A Al Omran and Christoph Treude. Choosing an NLP Library for Analyzing Software Documentation: A Systematic Literature Review and a Series of Experiments. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 187–197, Buenos Aires, Argentina, May 2017. IEEE. URL: `http://ieeexplore.ieee.org/document/7962368/`, `doi:10.1109/MSR.2017.42`.

[6] Miltiadis Allamanis and Charles Sutton. Why, when, and what: Analyzing Stack Overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 53–56, May 2013. ISSN: 2160-1852. `doi:10.1109/MSR.2013.6624004`.

[7] Catherine Ashcraft, Brad McLain, and Elizabeth Eger. *Women in tech: The facts*. National Center for Women & Technology (NCWIT), 2016.

[8] Andrew Begel and Beth Simon. Novice Software Developers, All over Again. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 3–14, New York, NY, USA, 2008. ACM. event-place: Sydney, Australia. URL: `http://doi.acm.org/10.1145/1404520.1404522`, `doi:10.1145/1404520.1404522`.

[9] Andrew Begel and Beth Simon. Struggles of new college graduates in their first software development job. In *ACM SIGCSE Bulletin*, volume 40, pages 226–230. ACM, 2008.

[10] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, September 2007. `doi:10.1109/TSE.2007.70725`.

[11] Maureen Biggers, Anne Brauer, and Tuba Yilmaz. Student perceptions of computer science: A retention study comparing graduating seniors with CS leavers. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 402–406, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1352135.1352274, doi:10.1145/1352135.1352274.

[12] Mike Bostock. D3.js - Data-Driven Documents, 2019. URL: https://d3js.org/.

[13] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

[14] Bureau of Labor Statistics. Software developers. Technical report, U.S. Department of Labor, Occupational Outlook Handbook, 2016-17 Edition, December 2015. URL: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm.

[15] Quinn Burke, Cinamon Bailey, Louise Ann Lyon, and Emily Greeen. Understanding the Software Development Industry's Perspective on Coding Boot Camps Versus Traditional 4-year Colleges. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 503–508, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3159450.3159485, doi:10.1145/3159450.3159485.

[16] Sally A. Carless and Jessica L. Arnup. A longitudinal study of the determinants and outcomes of career change. *Journal of Vocational Behavior*, 78(1):80–91, February 2011. URL: http://www.sciencedirect.com/science/article/pii/S000187911000151X, doi:10.1016/j.jvb.2010.09.002.

[17] Sapna Cheryan, Victoria C. Plaut, Caitlin Handron, and Lauren Hudson. The stereotypical computer scientist: Gendered media representations as a barrier to inclusion for women. *Sex roles*, 69(1-2):58–71, 2013. URL: http://link.springer.com/article/10.1007/s11199-013-0296-x.

[18] Pauline Rose Clance and Suzanne A. Imes. The imposter phenomenon in high achieving women: Dynamics and therapeutic intervention. *Psychotherapy: Theory, Research and Practice*, 15(3):241–247, 1978. URL: http://www.suzanneimes.com/wp-content/uploads/2012/09/Imposter-Phenomenon.pdf.

[19] Kevin A. Clarke and David M. Primo. *A Model Discipline: Political Science and the Logic of Representations.* OUP USA, February 2012. Google-Books-ID: d_voSSX2QgMC.

[20] Steven Clarke. Measuring API Usability. *Dr. Dobb's Journal*, Special Windows/.NET Supplement, May 2004. URL: `http://www.drdobbs.com/windows/measuring-api-usability/184405654`.

[21] Steven Clarke. What is an End User Software Engineerl. In Margaret H. Burnett, Gregor Engels, Brad A. Myers, and Gregg Rothermel, editors, *End-User Software Engineering*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL: `http://drops.dagstuhl.de/opus/volltexte/2007/1080`.

[22] Peter A. Cooper. Paradigm shifts in designed instruction: From behaviorism to cognitivism to constructivism. *Educational technology*, 33(5):12–19, 1993.

[23] Carl F. Craver. Structures of scientific theories. *The Blackwell guide to the philosophy of science*, pages 55–79, 2002.

[24] Barthelemy Dagenais and Martin P. Robillard. Recovering traceability links between an API and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 47–57, Zurich, June 2012. IEEE. URL: `http://ieeexplore.ieee.org/document/6227207/`, `doi:10.1109/ICSE.2012.6227207`.

[25] Aniket Dahotre, Vasanth Krishnamoorthy, Matt Corley, and Christopher Scaffidi. Using Intelligent Tutors to Enhance Student Learning of Application Programming Interfaces. *J. Comput. Sci. Coll.*, 27(1):195–201, October 2011. URL: `http://dl.acm.org/citation.cfm?id=2037151.2037190`.

[26] Frank Davidoff. Understanding contexts: how explanatory theories can help. *Implementation Science*, 14(1):23, March 2019. URL: `https://doi.org/10.1186/s13012-019-0872-8`, `doi:10.1186/s13012-019-0872-8`.

[27] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Using IR methods for labeling source code artifacts: Is it worthwhile? In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 193–202. IEEE, 2012.

[28] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Labeling source code with information retrieval methods: an empirical study. *Empirical Software Engineering*, 19(5):1383–1420, 2014.

[29] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 320–330, May 2009. `doi:10.1109/ICSE.2009.5070532`.

[30] Ekwa Duala-Ekoko and Martin P. Robillard. The information gathering strategies of API learners. Technical report, Technical report, TR-2010.6, School of Computer Science, McGill University, 2010. URL: `https://pdfs.semanticscholar.org/a7ff/4cc954744f761e8697be4e73aa25166a76c4.pdf`.

[31] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 266–276, Piscataway, NJ, USA, 2012. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2337223.2337255`.

[32] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*, pages 2491–2500, Toronto, Ontario, Canada, 2014. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=2556288.2556998`, `doi:10.1145/2556288.2556998`.

[33] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 73–82. IEEE, 2012.

[34] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[35] Thomas Fritz and Gail C. Murphy. Using Information Fragments to Answer the Questions Developers Ask. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM. event-place: Cape Town, South Africa. URL: `http://doi.acm.org/10.1145/1806799.1806828`, `doi:10.1145/1806799.1806828`.

[36] Elena L Glassman, Tianyi Zhang, Björn Hartmann, Miryung Kim, and UC Berkeley. Visualizing API Usage Examples at Scale. page 12, 2018.

[37] Elena L Glassman, Tianyi Zhang, Björn Hartmann, Miryung Kim, and UC Berkeley. Visualizing API Usage Examples at Scale. page 12, 2018.

[38] Mark Guzdial and Allison Elliott Tew. Imagineering inauthentic legitimate peripheral participation: An instructional design approach for motivating computing education. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 51–58, New York, NY, USA, 2006. ACM. URL: `http://doi.acm.org/10.1145/1151588.1151597`, `doi:10.1145/1151588.1151597`.

[39] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM, 2010.

[40] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 35–44. IEEE, 2010.

[41] John D. Hansen and Justin Reich. Democratizing education? Examining access and usage patterns in massive open online courses. *Science*, 350(6265):1245–1248, December 2015. 00000. URL: `http://www.sciencemag.org/content/350/6265/1245`, `doi:10.1126/science.aab3782`.

[42] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 3–12, October 2015. `doi:10.1109/VLHCC.2015.7356972`.

[43] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering*, page 11, Gothenburg, Sweden, 2018. `doi:10.1145/3180155.3180176`.

[44] Michael Hewner and Mark Guzdial. What Game Developers Look for in a New Graduate: Interviews and Surveys at One Game Company. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 275–279, New York, NY, USA, 2010. ACM. event-place: Milwaukee, Wisconsin, USA. URL: `http://doi.acm.org/10.1145/1734263.1734359`, `doi:10.1145/1734263.1734359`.

[45] Jenny M. Hoobler, Sandy J. Wayne, and Grace Lemmon. Bosses' Perceptions of Family-Work Conflict and Women's Promotability: Glass Ceiling Effects. *Academy of Management Journal*, 52(5):939–957, October 2009. URL: `https://journals.aom.org/doi/full/10.5465/amj.2009.44633700`, `doi:10.5465/amj.2009.44633700`.

[46] Jøri Gytre Horverak, Hege Høivik Bye, Gro Mjeldheim Sandal, and Ståle Pallesen. Managers' Evaluations of Immigrant Job Applicants: The Influence of Acculturation Strategy on Perceived Person-Organization Fit (P-O Fit) and Hiring Outcome. *Journal of Cross-Cultural Psychology*, 44(1):46–60, January 2013. URL: `https://doi.org/10.1177/0022022111430256`, `doi:10.1177/0022022111430256`.

[47] Jane Hsieh, Michael Xieyang Liu, Brad A. Myers, and Aniket Kittur. An Exploratory Study of Web Foraging to Understand and Support Programming Decisions. In *2018*

IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 305–306. IEEE, 2018.

[48] Mizuko Ito, Kris Gutiérrez, Sonia Livingstone, Bill Penuel, Jean Rhodes, Katie Salen, Juliet Schor, Julian Sefton-Green, and S. Craig Watkins. *Connected Learning*. BookBaby, Cork, 2013.

[49] He Jiang, Jingxuan Zhang, Xiaochen Li, Zhilei Ren, and David Lo. A more accurate model for finding tutorial segments explaining APIs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 157–167. IEEE, 2016.

[50] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. An unsupervised approach for discovering relevant tutorial fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering*, pages 38–48. IEEE Press, 2017.

[51] S. Jiang, A. Armaly, C. McMillan, Q. Zhi, and R. Metoyer. Docio: Documenting API Input/Output Examples. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 364–367, May 2017. `doi:10.1109/ICPC.2017.13`.

[52] Yasuyuki Kageyama. Openness to the unknown: The role of falsifiability in search of better knowledge. *Philosophy of the social sciences*, 33(1):100–121, 2003.

[53] Caitlin Kelleher and Michelle Ichinco. Towards a Model of API Learning. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 163–168, October 2019. ISSN: 1943-6092. `doi:10.1109/VLHCC.2019.8818850`.

[54] Robert E. Kelley. How to Be a Star Engineer. *IEEE Spectr.*, 36(10):51–58, October 1999. URL: `https://doi.org/10.1109/6.795608`, `doi:10.1109/6.795608`.

[55] Pekka Kilpelainen and Heikki Mannila. Ordered and Unordered Tree Inclusion. *SIAM Journal on Computing; Philadelphia*, 24(2):17, April 1995. URL: `http://search.proquest.com/docview/919612602/abstract/59AD938B1EB84520PQ/1`, `doi:http://dx.doi.org/10.1137/S0097539791218202`.

[56] J. Kim, S. Lee, S. Hwang, and S. Kim. Adding Examples into Java Documents. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 540–544, November 2009. `doi:10.1109/ASE.2009.39`.

[57] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353, May 2007. `doi:10.1109/ICSE.2007.45`.

[58] A. J. Ko and Y. Riche. The role of conceptual knowledge in API usability. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 173–176, September 2011. doi:10.1109/VLHCC.2011.6070395.

[59] A.J. Ko, B.A. Myers, and H.H. Aung. Six Learning Barriers in End-User Programming Systems. In *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 199–206, September 2004. doi:10.1109/VLHCC.2004.47.

[60] Amy J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1–2):41–84, February 2005. URL: http://www.sciencedirect.com/science/article/pii/S1045926X04000394, doi:10.1016/j.jvlc.2004.08.003.

[61] Amy J. Ko and Brad A. Myers. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1518701.1518942, doi:10.1145/1518701.1518942.

[62] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 361–370, 2007.

[63] Jean Lave and Etienne Wenger. *Situated learning: Legitimate peripheral participation.* Cambridge university press, 1991.

[64] Etienne P. LeBel, Randy J. McCarthy, Brian D. Earp, Malte Elson, and Wolf Vanpaemel. A unified framework to quantify the credibility of scientific findings. *Advances in Methods and Practices in Psychological Science*, 1(3):389–402, 2018.

[65] Colleen M. Lewis, Ruth E. Anderson, and Ken Yasuhara. “I Don't Code All Day”: Fitting in Computer Science When the Stereotypes Don't Fit. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 23–32, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2960310.2960332, doi:10.1145/2960310.2960332.

[66] Jing Li, Aixin Sun, and Zhenchang Xing. Learning to answer programming questions with software documentation through social context embedding. *Information Sciences*, 448-449:36–52, June 2018. URL: http://www.sciencedirect.com/science/article/pii/S0020025517303845, doi:10.1016/j.ins.2018.03.014.

[67] Paul Luo Li, Amy J. Ko, and Jiamin Zhu. What Makes a Great Software Engineer? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 700–710, Piscataway, NJ, USA, 2015. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2818754.2818839`.

[68] Wayne G. Lutters and Carolyn B. Seaman. Revealing actual documentation usage in software maintenance through war stories. *Information and Software Technology*, 49(6):576–587, June 2007. URL: `http://www.sciencedirect.com/science/article/pii/S0950584907000158`, `doi:10.1016/j.infsof.2007.02.013`.

[69] Louise Ann Lyon, Quinn Burke, Jill Denner, and Jim Bowring. Should your college computer science program partner with a coding boot camp? In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 712–712, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3017680.3022401`, `doi:10.1145/3017680.3022401`.

[70] Jane Margolis, Rachel Estrella, Joanna Goode, Jennifer Jellison Holme, and Kim Nao. *Stuck in the shallow end: Education, race, and computing.* MIT Press, 2010.

[71] Jane Margolis and Allan Fisher. *Unlocking the clubhouse: Women in computing.* MIT press, 2003.

[72] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290. ACM, 2014.

[73] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication*, page 0047281617721853, July 2017. URL: `http://dx.doi.org/10.1177/0047281617721853`, `doi:10.1177/0047281617721853`.

[74] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication*, 48(3):295–330, July 2018. URL: `https://doi.org/10.1177/0047281617721853`, `doi:10.1177/0047281617721853`.

[75] Leo A Meyerovich and Ariel Rabkin. Empirical Analysis of Programming Language Adoption. page 18.

[76] Laura Moreno. *Software documentation through automatic summarization of source code artifacts.* The University of Texas at Dallas, 2016.

[77] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 23–32. IEEE, 2013.

[78] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495. ACM, 2014.

[79] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Arena: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43(2):106–127, 2017.

[80] Mr.doob. mrdoob/three.js: JavaScript 3d library., February 2019. original-date: 2010-03-23T18:58:01Z. URL: `https://github.com/mrdoob/three.js`.

[81] Emerson Murphy-Hill, Caitlin Sadowski, Andrew Head, John Daughtry, Andrew Macvean, Ciera Jaspan, and Collin Winter. Discovering API Usability Problems at Scale. page 4, 2018.

[82] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming Q amp;A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, September 2012. `doi:10.1109/ICSM.2012.6405249`.

[83] National Center for Education Statistics. Bachelor's degrees conferred by postsecondary institutions, by field of study: Selected years, 1970-71 through 2016-17, 2018. URL: `https://nces.ed.gov/programs/digest/d18/tables/dt18_322.10.asp`.

[84] Najam Nazar, Yan Hu, and He Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, 2016.

[85] Jerome Neapolitan. Occupational change in mid-career: An exploratory investigation. *Journal of Vocational Behavior*, 16(2):212–225, April 1980. URL: `http://www.sciencedirect.com/science/article/pii/0001879180900524`, `doi:10.1016/0001-8791(80)90052-4`.

[86] Thanh Nguyen, Peter C. Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N. Nguyen. T2api: synthesizing API code usage templates from English texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1013–1017. ACM, 2016.

[87] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1595696.1595767`, `doi:10.1145/1595696.1595767`.

[88] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What Programmers Really Want: Results of a Needs Assessment for SDK Documentation. In *Proceedings of the 20th Annual International Conference on Computer Documentation*, SIGDOC '02, pages 133–141, New York, NY, USA, 2002. ACM. URL: `http://doi.acm.org/10.1145/584955.584976`, `doi:10.1145/584955.584976`.

[89] Stephen Oney and Joel Brandt. Codelets: linking interactive documentation and example code in the editor. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*, page 2697, Austin, Texas, USA, 2012. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=2207676.2208664`, `doi:10.1145/2207676.2208664`.

[90] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 63–72. IEEE, 2012.

[91] Chris Parnin and Christoph Treude. Measuring API documentation on the web. pages 25–30. ACM Press, 2011. URL: `http://portal.acm.org/citation.cfm?doid=1984701.1984706`, `doi:10.1145/1984701.1984706`.

[92] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd Documentation: Exploring the Coverage and the Dynamics of API Discussions on Stack Overflow. page 11, 2012.

[93] A. K. Peters and A. Pears. Engagement in Computer Science and IT – What! A Matter of Identity? In *2013 Learning and Teaching in Computing and Engineering*, pages 114–121, March 2013. `doi:10.1109/LaTiCE.2013.42`.

[94] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. Statistical learning of API fully qualified names in code snippets of online forums. In *Proceedings of the 40th International Conference on Software Engineering*, pages 632–642. ACM, 2018.

[95] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1295–1298. IEEE Press, 2013.

[96] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Prompter-Turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, 21(5):2190–2231, October 2016. URL: https://doi.org/10.1007/s10664-015-9397-1, doi:10.1007/s10664-015-9397-1.

[97] Lincoln Quillian, Devah Pager, Ole Hexel, and Arnfinn H. Midtbøen. Meta-analysis of field experiments shows no change in racial discrimination in hiring over time. *Proceedings of the National Academy of Sciences*, 114(41):10870–10875, October 2017. URL: https://www.pnas.org/content/114/41/10870, doi:10.1073/pnas.1706255114.

[98] Quincy Larson. We asked 15,000 people who they are, and how they're learning to code, May 2016. URL: https://medium.freecodecamp.com/we-asked-15-000-people-who-they-are-and-how-theyre-learning-to-code-4104e29b2781.

[99] Course Report. 2016 Coding Bootcamp Market Size Study, June 2016. URL: https://www.coursereport.com/reports/2016-coding-bootcamp-market-size-research.

[100] Course Report. 2016 Course Report alumni outcomes & demographics study, September 2016. URL: https://www.coursereport.com/reports/2016-coding-bootcamp-job-placement-demographics-report.

[101] Course Report. 2018 Coding Bootcamp Alumni Outcomes & Demographics Report, December 2018. URL: https://www.coursereport.com/reports/coding-bootcamp-job-placement-2018.

[102] Course Report. 2018 coding bootcamp market size study, August 2018. URL: https://www.coursereport.com/reports/2018-coding-bootcamp-market-size-research.

[103] Susan R. Rhodes and Mildred Doering. An integrated model of career change. *Academy of Management Review*, 8(4):631–639, October 1983. URL: http://amr.aom.org/content/8/4/631, doi:10.5465/AMR.1983.4284666.

[104] Peter C. Rigby and Martin P. Robillard. Discovering essential code elements in informal documentation. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 832–841. IEEE, 2013.

[105] Neil Robertson and P. D Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, September 1986. URL: http://

www.sciencedirect.com/science/article/pii/0196677486900234, doi:10.1016/0196-6774(86)90023-4.

[106] M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, November 2009. doi:10.1109/MS.2009.193.

[107] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013. doi:10.1109/TSE.2012.63.

[108] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 11–20. ACM, 2005.

[109] Martin P. Robillard and Yam B. Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[110] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, December 2011. URL: https://link.springer.com/article/10.1007/s10664-010-9150-8, doi:10.1007/s10664-010-9150-8.

[111] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. On-demand developer documentation. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 479–483. IEEE, 2017.

[112] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, and others. On-Demand Developer Documentation. *Software Maintenance and Evolution (ICSME)*, 2017. URL: http://www.inf.usi.ch/lanza/Downloads/Robi2017a.pdf.

[113] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 247–258, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2984511.2984544, doi:10.1145/2984511.2984544.

[114] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009. URL: http://www.sciencedirect.com/science/article/pii/S0167642309000367, doi:10.1016/j.scico.2009.02.007.

[115] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 191–201, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2786805.2786855, doi:10.1145/2786805.2786855.

[116] Carsten Schulte and Maria Knobelsdorf. Attitudes towards computer science-computing experiences as a starting point and barrier to computer science. In *Proceedings of the Third International Workshop on Computing Education Research*, ICER '07, pages 27–38, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1288580.1288585, doi:10.1145/1288580.1288585.

[117] Araba Sey and Maria Garrido. Coding bootcamps: A strategy for youth employment in developing countries. Research Report, Technology & Social Change Group, University of Washington, May 2016. URL: http://tascha.uw.edu/publications/coding-bootcamps-a-strategy-for-youth-employment-in-developing-countries/.

[118] F. Shull, F. Lanubile, and V. R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, 26(11):1101–1118, November 2000. doi:10.1109/32.881720.

[119] J. Sillito and A. Begel. App-directed learning: An exploratory study. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 81–84, May 2013. doi:10.1109/CHASE.2013.6614736.

[120] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34. ACM, 2006.

[121] Roslyn Smart and Candida Peterson. Super's Career Stages and the Decision to Change Careers. *Journal of Vocational Behavior*, 51(3):358–374, December 1997. URL: http://www.sciencedirect.com/science/article/pii/S0001879196915444, doi:10.1006/jvbe.1996.1544.

[122] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984. doi:10.1109/TSE.1984.5010283.

[123] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[124] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 101–110. ACM, 2011.

[125] Christian Sturm, Alice Oh, Sebastian Linxen, Jose Abdelnour Nocera, Susan Dray, and Katharina Reinecke. How WEIRD is HCI?: Extending HCI Principles to other Countries and Cultures. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2425–2428. ACM, 2015. 00000. URL: `http://dl.acm.org/citation.cfm?id=2702656`.

[126] J. Stylos and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 195–202, September 2006. `doi:10.1109/VLHCC.2006.32`.

[127] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 643–652. ACM, 2014.

[128] Patrick Suppes. What is a Scientific Theory? *Philosophy of Science Today*, pages 55–67, 1967.

[129] K. Thayer. Using Program Analysis to Improve API Learnability. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 271–272, October 2018. `doi:10.1109/VLHCC.2018.8506583`.

[130] K. Thayer. Using Program Analysis to Improve API Learnability. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 271–272, October 2018. `doi:10.1109/VLHCC.2018.8506583`.

[131] Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. A theory of robust api knowledge. *Under review*, 2020.

[132] Kyle Thayer and Amy J. Ko. Barriers Faced by Coding Bootcamp Students. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 245–253, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3105726.3106176`, `doi:10.1145/3105726.3106176`.

[133] TIOBE. TIOBE index | TIOBE - the software quality company, May 2018. URL: `https://www.tiobe.com/tiobe-index/`.

[134] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from Stack Overflow. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 392–403. IEEE, 2016.

[135] Yu-Cheng Tu, Gillian Dobbie, Ian Warren, Andrew Meads, and Cameron Grout. An Experience Report on a Boot-Camp Style Programming Course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 509–514, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3159450.3159541`, `doi:10.1145/3159450.3159541`.

[136] G. Uddin and M. P. Robillard. How API Documentation Fails. *IEEE Software*, 32(4):68–75, July 2015. `doi:10.1109/MS.2014.80`.

[137] Gias Uddin and Martin P. Robillard. Resolving API mentions in informal documents. *arXiv preprint arXiv:1709.02396*, 2017.

[138] U.S. Bureau of Labor Statistics. Most New Jobs : Occupational Outlook Handbook, April 2019. URL: `https://www.bls.gov/ooh/most-new-jobs.htm`.

[139] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, August 1995. URL: `http://ieeexplore.ieee.org/document/402076/`, `doi:10.1109/2.402076`.

[140] Robert Bennett Watson. *The effect of visual design and information content on readers' assessments of API reference topics.* PhD Thesis, 2015.

[141] Etienne Wenger. *Communities of practice: Learning, meaning, and identity.* Cambridge university press, 1998.

[142] Etienne Wenger-Trayner and Beverly Wenger-Trayner. Communities of practice a brief introduction, April 2015. URL: `http://wenger-trayner.com/wp-content/uploads/2015/04/07-Brief-introduction-to-communities-of-practice.pdf`.

[143] Judy Williams. Constructing a new professional identity: Career change into teaching. *Teaching and Teacher Education*, 26(3):639–647, April 2010. URL: `http://www.sciencedirect.com/science/article/pii/S0742051X09001966`, `doi:10.1016/j.tate.2009.09.016`.

[144] Brenda Cantwell Wilson and Sharon Shrock. Contributing to success in an introductory computer science course: A study of twelve factors. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 184–188, New York, NY, USA, 2001. ACM. URL: `http://doi.acm.org/10.1145/364447.364581`, `doi:10.1145/364447.364581`.

[145] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567. IEEE, 2013.

[146] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185, 2017.

[147] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 419–429, Washington, DC, USA, 2003. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=776816.776867.

[148] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic Parameter Recommendation for Practical API Usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2337223.2337321.

[149] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP 2009 – Object-Oriented Programming*, pages 318–343. Springer, Berlin, Heidelberg, July 2009. URL: https://link.springer.com/chapter/10.1007/978-3-642-03013-0_15, doi:10.1007/978-3-642-03013-0_15.

Appendix A
BOOTCAMP STUDY INTERVIEW PROTOCOL

A.1 Upfront Disclosure:

I am conducting a research study for the University of Washington to understand why people are participating in coding bootcamps and how their views about coding bootcamps change as a result of their participation. This information may be useful to people considering participating in coding bootcamps and to the coding bootcamps themselves, but we will not share your name or any identifiable information publicly, or with the bootcamps.

Since you have participated in a coding bootcamp, I wanted to ask you about your reasons for participating. I'll be interviewing multiple people, so, in order to make sure that I'm asking the questions consistently, I'm going to be reading the questions from a script. There are several sets of questions and this interview should take approximately one hour.

I am going to record this interview to ensure that I get everything that you say, and also so I can focus on asking questions instead of taking notes. The recording will be erased after I transcribe the data, and all personally identifiable information, such as people's names, will be removed. I can share findings with you once I have written up the results.

I'm going to be asking a lot of "why" and "how" questions so I can better understand why something is important; the more you can elaborate on, the better. Your participation is voluntary and optional. You do not have to answer any question that you don't want to and can stop at any time. You can also come talk to me after this is finished if you have any questions or concerns.

Do you have any questions before we get started?

[Address questions, wait for confirmation]

A.2 Interview

[start recording]

My first set of questions is on what which coding bootcamps you've participated in and your background with programming:

- Which coding bootcamp or bootcamps have you participated in?
- How much time have you spent at the coding bootcamp(s) so far and how much time is left?
- What size were your classes in the coding bootcamp(s)?
- Besides coding bootcamps, how else have you learned programming skills?

My second set of questions is on how you made decisions about participating in [latest bootcamp].

- What were your goals in attending this coding bootcamp?
- Why did you choose [latest bootcamp] in particular?
- How did you decide to set the goal(s) of [goals]?
- How do you see [latest bootcamp] as fitting in with [goals]?
- What sacrifices or obstacles did you face, if any, in choosing to learn programming skills in general?
- What sacrifices or obstacles did you face, if any, in choosing to attend [latest coding bootcamp] in particular?
- Is there anything else in your story of deciding to participate in [latest bootcamp] that you think is important?

My third set of questions is on how your views on coding bootcamps have changed and how your goals have changed since starting a coding bootcamp.

- How has attending [latest bootcamp] affected your confidence in your ability to achieve your goals of [goals]?
- How has attending [latest bootcamp] affected your desire to achieve the goals of [goals]?
- In what ways do you feel like [latest bootcamp] is or is not giving you the necessary skills and tools to achieve your goals of [goals]?
- [If there were skills and tools the bootcamp was not giving them]

  What, if anything, are you doing or have you done [to get the skills and tools they mentioned]?

My fourth set of questions are on how your experience compares with your perception others' experiences in [latest bootcamp].

- Think of the other people you've met at [latest bootcamp]. How do you think your experiences have been similar or different from theirs?
- While attending [latest bootcamp], in what ways have you felt like you did or did not fit in with the other students and teachers?
- How do you feel that your sense of belonging at [latest bootcamp] would be similar or different at [your goal work environment (if there is one)]?
- If you could go back in time and give yourself advice about participating in [latest bootcamp], what would you say?
- [If not already answered]

  If you could go back in time, would you still choose to attend [latest bootcamp]?

- If you could go back in time and give yourself advice about your goals of [goals], what would you say?

- I'm also interested in how identity factors such as race, ethnicity, age, sexual orientation and/or gender identity affect participation in coding bootcamps.

- *I'm curious how your feel your race, ethnicity, sexual orientation and/or gender identity have affected your experience at [latest coding bootcamp] (if they have). Can you comment on that?*

  - [If they do not give personal experience]

    I'm particularly interested in your lived experience. How your feel your race, ethnicity, sexual orientation and/or gender identity have affected your lived experience at [latest coding bootcamp] (if they have)

  - It would also help me in writing my paper if you could provide some demographic information for me, in particular, race, ethnicity and age and gender. Do you mind telling me those?

Those are all my main questions.

- Do you have any final thoughts on your decision to participate in a coding bootcamp?

My one last question for you is:

- Who else do you know who has participated in a coding bootcamp who might be willing to be interviewed by me?

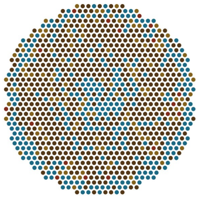
Appendix B
API KNOWLEDGE STUDY 1 TASK DESCRIPTIONS

d3.js

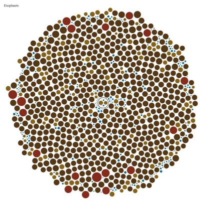
Exoplanet Visualization

Start/example code sources: https://bl.ocks.org/mbostock/3007180 https://beta.observablehq.com/@mbostock/d3-circle-packing

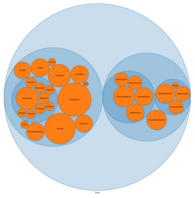
Start Output



Goal Output



Example Code Output



1) Add title to the top-left that says "Exoplanets" (see Final Graph below)
1. Use the svg variable
2. Call append()
3. Call text()
4. Call attr() for dx,dy
5. Position correctly

2) Make the circles have an area proportional with the square of the planet radius. Make sure the circles stay close together (reduce big spaces).
1. Change attr("r")
2. Use d.r or d.data.radius in some way
3. To d.r or d.data.radius (allow for constant multipliers)
4. Modify sum()
5. To radius * radius

3) Add the planets of our solar system to the graph (the Array.concat function may help). It is difficult to tell if this worked, so if you think you have it, go to the next task.
1. Concat planets to array

4) Put planets of our solar system in a circle, separated from the rest of the planets. Make the circle surrounding the planets of our solar system not be filled in (you can use the provided "hollow-circle" class in the css).
1. Make separate child in hierarchy for our solar system
2. Use class for circle

5) Sort planets by distance to the sun. That is, planets in the middle are close to the sun, planets on the outside are far from the sun.
1. Use sort function on D3 hierarchy
2. Correctly perform sort (including handling children and NANs)

Task

This text was shown to participants.

Subtasks

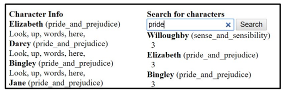
These were not shown to participants. They were used by researchers to measure progress.

Natural

Book Character Word Association

Example code source: https://github.com/NaturalNode/natural

Start Output

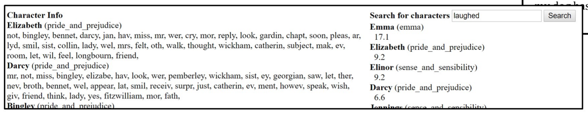


Character Info
Elizabeth (pride_and_prejudice)
Look, up, words, here,
Darcy (pride_and_prejudice)
Look, up, words, here,
Bingley (pride_and_prejudice)
Look, up, words, here,
Jane (pride_and_prejudice)

Search for characters
pride

Willoughby (sense_and_sensibility)
3
Elizabeth (pride_and_prejudice)
3
Bingley (pride_and_prejudice)
3

Goal Output

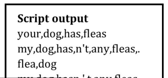


Character Info
Elizabeth (pride_and_prejudice)
not, bingley, bennet, darcy, jan, hav, miss, mr, wer, cry, mor, reply, look, gardin, chapt, soon, pleas, ar, lyd, smil, sist, collin, lady, wel, mrs, felt, oth, walk, thought, wickham, catherin, subject, mak, ev, room, let, wil, feel, longbourn, friend,
Darcy (pride_and_prejudice)
mr, not, miss, bingley, elizabe, hav, look, wer, pemberley, wickham, sist, ey, georgian, saw, let, ther, nev, broth, bennet, wel, appear, lat, smil, receiv, surpr, just, catherin, ev, ment, howev, speak, wish, giv, friend, think, lady, yes, fitzwilliam, mor, fath,
Bingley (pride_and_prejudice)

Search for characters laughed

Emma (emma)
17.1
Elizabeth (pride_and_prejudice)
9.2
Elinor (sense_and_sensibility)
9.2
Darcy (pride_and_prejudice)
6.6
Jennings (sense_and_sensibility)

Example Code Output



Script output
your,dog,has,fleas
my,dog,has,n't,any,fleas,.
flea,dog
my,dog,hasn,',t,any,fleas,.

1) getWordsNextToCharacterName currently returns words that occur up to two before and two after a character name, excluding the name itself (i.e., "word1 word2 characterName word1 word2"). You want to modify it to find words that occur up to six words before or six words after a character name. Check the results in the debugging console.
*1. Modify the trigrams() line
2. Use the ngrams() function
3. Use ngrams(, 7)

2) You want to find out which words commonly appear near a character's name (relative to words near other characters' names). For example: "she" often appears near the name Elizabeth, and more so than near other character names. Replace the code in findWordsForCharacters that sets ["Look", "up", "words", "here"] with code that finds those commonly appearing words (you may have to modify other code as well). To do this, use the Natural library to group the strings returned by getWordsNextToCharacterName and find those words.
*1. Use tfidf in some way
2. Add documents
3. Add ngrams from each character as a document
4. Call listterms()
5. Of character i
6. Try to print it
7. Using charwords[j].term

3) You want to be able to search for characters that best fit words. Use your work from task 2 and the Natural library to find how commonly the search words appear near a character's name (relative to other characters). Fill in the match value in the search function. Then search should work like this:
1. Make tfidf global (or redo work)
2. Replace the match value
3. With result of tfidf.tfidf() function
4. Passing it (searchTerms, i)

4) You notice some words found aren't ones you are interested in (like "she" and "not"). Use the Natural library to determine the type of word found based on the context of the group of seven words you found it with. Then ignore the following types of words:
- **Coordinating Conjunctions, like "nor"**
- **Determiners, like "the"**
- **Preposition or subordinating conjunctions, like "towards"**
- **Personal pronouns, like "himself"**
- **Possessive pronouns, like "our"**

1. Set up rules / lex / tagger
2. Call tagger.tag
3. Of ngram words
4. Do for loop over tags
5. Get the part of speech value
6. Check if it is not some parts of speech
7. Specifically PRP PRP$ IN DT CC

5) You notice that searching for "laugh" and "laughed" brings up different results, but you think those should be combined. Use the Natural library to combine related words like "laugh" and "laughed". Make sure search works with this.
1. Call the stem() function
2. Stem the ngram words before tfidf add documents
3. Stem words user entered before searching

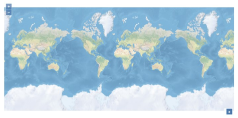
Figure B.1: Task sets for d3.js and Natural APIs. Shows the output of the starter code participants were given, the final output they were asked to produce, and the output of the example code they were given as a reference. Task text participants were given are in bold (intermediate goal images were shown with some tasks, but are not shown here). The subtasks that were used by researchers to measure progress are in light typeface between the tasks. Substasks used as bottlenecks (section 3.4.2) are highlighted with a star.

OpenLayers

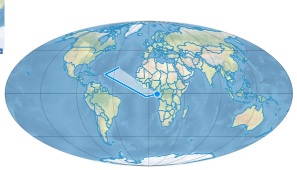
Interactive Map

Start/example code source: https://openlayers.org/en/latest/examples/

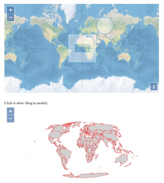
Start Output



Goal Output



Example Code Output



1) Change your map to be a rounded map using the sphereMollweideProjection variable already defined. If it works, you should see ridges along the top of the map.
1. Set projection to sphereMollweideProjection
2. Set view projection to sphereMollweideProjection

2) Add lines representing latitude and longitude to the map.
*1. Use graticule
2. Correctly store map to variable and use in graticule call

3) Add outlines to the countries from: https://openlayers.org/en/v4.6.4/examples/data/geojson/countries-110m.geojson (url in example code). Make sure you don't cover the countries with any color, but just have the outlines. Those outlines should be color #3399CC and have a width of 1.5.
1. Create source vector
2. With url and GeoJson
3. Create vector layer
4. With source vector (make sure vector is defined first so this isn't undefined)
5. Set style
6. Fill correctly (or have no fill style)
7. Stroke correctly (color and width)
8. Add vector layer to map

4) Cover the areas outside the main globe (marked with latitude/longitude lines) with white so that you just see the main oval of the earth. Don't worry about covering the outside area perfectly, just get it close.
1. Create linear rings
2. Circle
3. Resized to ellipse
4. Rectangle
5. Create polygon
6. With Rectangle then circle
7. Sized correctly
8. Create vector source
9. With polygon
10. Create vector layer
11. With vector source
12. Style it white
13. Add layer to map

5) Allow users to draw shapes and modify them. Don't let them modify the country borders or any other feature. (Use OpenLayers default colors for the drawn shapes).
1. New layer
2. Interaction - modify
3. Interaction - draw

6) To aide with drawing, make it so the map makes it easier to draw points along the edges and corners of countries and previously drawn shapes.
1. Add snap to new layer
2. Add snap to country layer

Task
This text was shown to participants.

Subtasks
These were not shown to participants. They were used by researchers to measure progress.

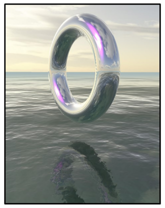
ThreeJS

Spinning Loading Ring

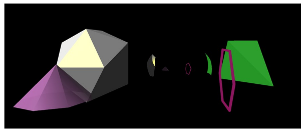
Start/example code sources: https://threejs.org/examples/ https://threejs.org/examples/#webgl_shaders_ocean

Start Output

Goal Output



Example Code Output



1) Add this shape. Make it sized with an overall diameter of 40, the tube part with a diameter of about 10, and position the bottom 10 above the water. It should look smooth and circular. See the next page for clearer views of the shape.
1. Create any geometry
2. Create material (give point if they make a mesh with no material)
3. Create mesh from the geometry and material
4. Add to scene
*5. TorusGeometry()
6. Sized correctly (20,5,...)
*7. Smooth (fourth parameter)
8. Position changed
9. Positioned correctly (position.y = 30)

2) Make the shape be gray (hex color: 444444) and have a purple shine (hex color: 991199).
1. Phong Material
2. setting emissive instead of color
3. Color correct (in emissive)
4. Specular set (Shininess set optionally)
5. To correct color

3) Make the shape appear perfectly reflective (note: the reflections can be hard to see if your shape is too dark, so make it white [hex color: ffffff] and uniformly shaded). Make sure this reflection work for both the sky and ocean.
1. Color white
2. That white color should be set in emmisive
3. Reflect camera
4. Phong material
5. With env map
6. Set cube reflection mapping
7. Set camera position to match torus
8. Update camera in animate

4) Make the doughnut spin around it's vertical axis to indicate that the website is loading.
1. Get current time
2. Change object rotation based on current time
3. Specifically change rotation.y

Figure B.2: Task sets for OpenLayers and ThreeJs APIs. Shows the output of the starter code participants were given, the final output they were asked to produce, and the output of the example code they were given as a reference. Task text participants were given are in bold (intermediate goal images were shown with some tasks, but are not shown here). The subtasks that were used by researchers to measure progress are in light typeface between the tasks. Substasks used as bottlenecks (section 3.4.2) are highlighted with a star.

Appendix C

API USAGE TEMPLATE G-AST EXPERT REVIEWS

randIc	API Usage Pattern G-AST	P1 responses	P2 responses	stated purpose	Problem Codes	Notes:

1	var «?1» = 500;
	var «?2» = «...».attr(«...», «?1»);
	d3.max(«?3», «...»)
	«...»("class", «...»)
		Recognize? Yes
		purpose:
		1: assinging variable class
		2: settting some attribute using 1 (500)
		[line 3] <- finding max
		[line 4] <- assigning
		problem:
			Recognize? [blank]
			purpose:
			The first 2 lines set attributes for an element. The 3rd line computes max of something, but seems unrelated to the last line.
			problem:
			var <<?2>> is not reffered, nor is the result of d3.max. What does the last line do?
				Both	Abstract-Call (P2)
					Disconnected (P2)
					Incomplete (P2)	

2	var «?1» = 300;
	var svg = d3.select("svg").attr("width", «?2»).attr("height", «?1»);
	svg.selectAll(«...»).data(«?3»).enter().append(«...»).attr(
	"y", function «...»(d){
	}
	)
		Recognize? Yes
		purpose:
		Set svg width and height. Then bind data and create children for that svg parent. Set y attr for all the children using the data.
		problem:
			Recognize? Yes
			purpose:
			It sets the width and height of an svg element and starts drawing visualizaion on it using data.
			problem:
			<<?2>> is not defined here...
				Both	Incomplete (P2)	

3	d3.«?2»().domain(«?1»).range	Recognize? Yes purpose: Setting the domain and range of a scale/axis Because range doesn't have () it looks like a read only field. Althrough in d3 if you want to read you'd still write .range() problem:	Recognize? Yes purpose: It sets axis scale problem: range should also have a ()	Both	Incorrect (P1, P2)	
4	d3.select("body").append("svg").attr(«...», «...»).attr(«...», «?1»).attr(«...», «...»)	Recognize? Yes purpose: Setting three attr for svg problem:	Recognize? Yes purpose: It creates an svg and append it to the body element, and modify the attributes of svg problem: Why is <<?1>> special?	Both	Spec or Q (P2)	

5	var «?1» = d3.select(«...»).append("div").attr("class", "tooltip").style(«...», «...»).style(«...», «...»);
	function (){
	}
		Recognize? Yes
		purpose:
		Creating a tooltip div and setting its css style.
		Not sure what function() is doing here because the d3 line has already created the tooltip
		problem:
			Recognize? Yes
			purpose:
			It creates a div and add it to an element then modify the style and add a class "tooltip" to the div
			problem:
			function(){} is not doing anything
				Both	Abstract-Fn (P1, P2)	

6	function (data){
	  d3.«?2»(«?1», «...»)
	  «...»(data)
	}
		Recognize? Yes
		purpose:
		The data line looks like data binding
		problem:
		But I don't really know because of <<...>>
			Recognize? No
			purpose:
			
			problem:
			(A) too abstract
				P1	Abstract-Overall (P1, P2)	

7	«?1».nodes(graph.nodes) «...»(graph.«?2») d3.«?3»(«...»)	Recognize? Yes purpose: I'm not familiar with network and nodes but looks like line #1 is setting the nodes of a network using nodes from an existing graph. Not quite sure. problem:	Recognize? No purpose: problem: I haven't used graph before	Neither	Unfamiliar (P1, P2)	
8	var svg = d3.select("svg");	Recognize? Yes purpose: Select the first svg d3 encounters problem:	Recognize? Yes purpose: It selects and svg element problem: It's too specific and not doing anything useful...	Both	Incomplete (P2)	

9	var «?1» = d3.select(«...»); «?1».attr(«...») «?1».attr(«...»)	Recognize? Yes purpose: Select something and set its attributes problem:	Recognize? Yes purpose: It sets attribute for an element problem:	Both		

10	var svg = d3.select("svg");
	svg.attr(«...»)
	svg.attr(«...»)
		Recognize? Yes
		purpose:
		Select the 1st svg and set its attributes
		problem:
			Recognize? Yes
			purpose:
			It sets attributes of an svg element
			problem:
				Both		

11	d3.«?1»(x) x(«...»)	Recognize? Yes purpose: x looks like a callback by usually callbacks wouldn't be called like x problem:	Recognize? No purpose: problem: It is too abstract and perhaps incorrect	Neither	Abstract-Overall (P2) Incorrect (P1, P2)	Source file (looks ok): x = d3.scaleLinear(); [...] d3.axisBottom(x) [...] .attr('cx', d => x(d[xVal])) Reference file (unrelated code): d3.pointRadial(e.x, e.y); [...] d3.linkVertical().x(function(d) { return d.x; })
12	d3.«?1»(«...»).then(«...»)	Recognize? Yes purpose: Promise problem:	Recognize? No purpose: problem: I don't recognize "then". Is it a promise pattern?	P1	Unfamiliar (P2)	

13	var «?1» = d3.scaleOrdinal().range([]);
	«...»(«...», «...», function (error, data){
	
	«...».selectAll(«...»).data(«...»(data)).enter().append(«...
	»).attr("fill", function (d) {
	    return «?1»(d.«?2»);
	  }
	)
	}
	)
		Recognize? Yes
		purpose:
		Create an ordinal scale, bind data, create marks for the new data.
		problem:
			Recognize? Yes
			purpose:
			It creates a cateorical scale. Then in an asynch function callback, it binds data to some elements and sets the fill color using the categorical scale.
			problem:
				Both		

14	var svg = d3.select(«...»).append("svg").attr("width", «?2»).attr("height", «?1»);
	«...»(«...», function (){
	}
	)
		Recognize? Yes
		purpose:
		Select svg and set attr. Not sure what the function() is doing below/
		problem:
			Recognize? Yes
			purpose:
			The first line adds a svg element to something and sets the width and height of the svg
			problem:
			I don't recognize the second line. It's too abstract.
				Both	Abstract-Fn (P1, P2)
					Abstract-Call (P1, P2)	

15	«?4».append(«...»).attr(«...», «...») var «?1» = svg.selectAll(«...»).data(«?2»); d3.«?3»(«?1»)	Recognize? Yes purpose: 1 is binding data(2) not sure what this line [3] does problem:	Recognize? Yes purpose: It adds an element to something and sets its attributes. It then binds data to something problem: I don't know what the 3rd line is doing: it's too abstract.	Both	Abstract-Call(P1, P2)	

16	«...»(function (){
	  d3.«?1»(«...»).append(«...»).attr(«...», «...»)
	}
	)
		Recognize? Yes
		purpose:
		I understand the d3 line but not the outer <<..>>(). So it's a function that takes a function as an input.
		problem:
			Recognize? Yes
			purpose:
			It creates and modifies the attributes of an element inside a function
			problem:
			I don't know what the function is for.
				P2	Abstract-Fn (P1, P2)	

17	d3.max(«?1», «...»)
	data.«?2»(function d(){
	}
	)
		Recognize? Yes
		purpose:
		take max.
		data manipulation
		problem:
			Recognize? No
			purpose:
			
			problem:
			I don't know if data can have properties
				P1	Spec or Q (P2)	Initial makes sense:
						   data.map(data.map(d => d.product))
						
						reference does not:
						svg.selectAll("rect").data(dataset).enter().append("rect").attr("y", function(d) {return svgHeight - yScale(d) })

18	var «?1» = d3.«?4»().range([]);
	function «...»(«?2», data){
	  d3.max(data, function (d) {
	  }
	)
	  «?1».domain(«...»)
	  «?1»(d.«?3»)
	}
		Recognize? Yes
		purpose:
		empty the scale range. The function is [sic]
		[line 3] taking data max
		[line 6] can't tell if it's related to data
		[line 7] is this [d] data? Not sure
		problem:
			Recognize? Yes
			purpose:
			It computes and sets axis scale for something
			problem:
			The last line may be unrelated and too abstract
				Both	Abstract-Call (P1, P2)
					Disconnected (P1, P2)
						source: d is from the data, reference from separate json data

19	const svg = d3.select(«...»)	Recognize? Yes purpose: select an svg problem:	Recognize? Yes purpose: It selects some element, possibly an svg problem: It's not doing anything useful. Svg should appear inside select	Both	Incomplete (P2) Spec or Q (P2)	

20	d3.csv("data.csv", function (error, data){
	  data.«?1»(function (d) {
	  }
	)
	  «...»(data, «...»)
	  «...»(data)
	}
	)
		Recognize? Yes
		purpose:
		Reading csv data. Some data manipulation / cleaning / processing.
		problem:
			Recognize? Yes
			purpose:
			It loads a csv file and does something after the data is loaded
			problem:
			the * lines [5 and 6] are too abstract
				Both		

21	width height d3.«?3»(«...»).append(«...»).attr(«...», «?2»).attr(«...», «?1»)	Recognize? Yes purpose: I guess the 3rd line could be setting width and height of the selected element. Although the code itself doesn't say that problem:	Recognize? No purpose: problem: The * lines [1 and 2] appear incorrect	P1[?]	Incorrect (P1) Disconnected (P1) Spec or Q (P1)	
22	d3.«?1»("body").append("svg").attr("width", «...»).attr("height", «...»)	Recognize? Yes purpose: Add an svg under HTML body and set its width & height problem:	Recognize? Yes purpose: It creates an svg, sets its width and height, and addds it to body. problem: <<?1>> is almost always "select"?	Both	Spec or Q (P2)	One is "select" one is "selectAll"
23	d3.select(«...»).append("svg")	Recognize? Yes purpose: Add an svg under some parent selected problem:	Recognize? Yes purpose: It creates an svg element and appends it to something problem:	Both		

24	d3.max(«?1», «...»)
	«?2»(data)
	function «...»(data){
	}
		Recognize? Yes
		purpose:
		take max from 1
		process data using 2
		3 is some data processing function
		problem:
			Recognize? No
			purpose:
			
			problem:
			The last two lines are too abstract
				P1	Abstract-Fn (P2)
					Abstract-Call (P2)	

25	d3.max(data, function d(){
	  return d.«?1»;
	}
	)
	function (){
	}
		Recognize? Yes
		purpose:
		take max in data based on attribute <11>
		problem:
		[line 1] I think d should be inside the (). Like an argument
			Recognize? Yes
			purpose:
			It computes the max of a data field
			problem:
			* [function at end] is not doing anything (unrelated?)
				Both	Abstract-Fn (P1, P2)
					Incorrect (P1)
						Location of D is indeed incorrect. Perhaps error in parsing fat arrow: "d3.max(data, d => d.tons);"

26	function («?1»){
	d3.«?4»(«...»).append(«...»).attr(«...», «?3»).attr(«...», «?2»)
	}
		Recognize? Yes
		purpose:
		a function that creates new elements and sets attributes
		[note on <<4>>] select(...) I automatically think
		problem:
			Recognize? Yes
			purpose:
			It creates an element and sets its attributes. Inside a function
			problem:
			Why is function necessary? What does it do?
				Both	Abstract-Fn (P2)
					Spec or Q (P1)
						Spec. correct for source, in ref it is d3.csv with of g.append() appearing inside

27	var projection = d3.«?1»();
	projection
	projection
	projection([])
			Recognize? No
			purpose:
			problem:
			It doesn't tell us what projection is. Also * [lines 2 and 3] seems incorrect
				NotP2	Abstract-Overall (P2)
					Incorrect (P2)	

28	let «?1» = d3.scaleLinear().range([])		Recognize? Yes purpose: It creates a linear scale and sets its range problem:	P2		

29	var «?1» = d3.«?2»().range([]);
	d3.axisBottom(«?1»)
			Recognize? Yes
			purpose:
			It creates a scale and use it on the x-axis that appear at the bottom
			problem:
			<<?2>> is scale?
				P2	Spec or Q (P2)	in examples is d3.scaleTime or d3.scaleLinear

30	{
	«...»(function (d) {
	return d.«?2»;
	}
	)
	d3.«?1»(data, function (d) {
	return d.«?3»;
	}
	)
	}
			Recognize? No
			purpose:
			problem:
			The outer "{}" is unnecessary
			It's too abstract
				NotP2	Abstract-Overall (P2)
					Other (P2)	

31	width d3.select(«...»).append(«...»).attr(«...», «?1») width «...».attr(«...», width).attr(«...», «...»).attr(«...», «...»)		Recognize? Yes purpose: It creates and element, sets its attribute and and appends it to something. It then modifies the attribute of some element problem: * [lines 1,3] seems incorrect	P2	Incorrect	
32	d3.«?1»().range([])		Recognize? Yes purpose: It sets the range of something problem:	P2		

33	var «?1» = d3.«?2»().range([]);
	«...»(function (d) {
	}
	).y(function (d){
	return «?1»(d.«?3»);
	}
	)
			Recognize? Yes
			purpose:
			Maybe it sets the axis scale of the y-axis? I'm not sure.
			problem:
			* [lines 2-3] is unrelated or too abstract
				P2(?)	Abstract-Call (P2)
					Disconnected (P2)	

34	var svg = d3.select(«...»).append("svg").attr("width", «?2»).attr("height", «?1»); svg.«?3»(«...»).attr(«...», «...»)		Recognize? Yes purpose: It creates an svg and sets its width and height. It futher modifies svg. problem: * ["<<?3>(<<...>>)"] is too abstract	P2	Abstract-Call (P2)	

35	var «?1» = d3.«?3»().range([]);
	function «...»(«?2», data){
	  «?1».domain([d3.max(data, function (d) {
	    }
	)])
	}
			Recognize? Yes
			purpose:
			It sets the domain and range of something. Domain is computer using the max of a data field.
			problem:
				P2		

36	«...»(«...», function (){
	  d3.select(«...»)
	}
	)
			Recognize? Yes
			purpose:
			??
			problem:
			It's too abstract
				NotP2	Abstract-Overall (P2)	

37	var svg = d3.select("svg");
	svg.append("g")
			Recognize? Yes
			purpose:
			It selects an svg and append a "g" element to it
			problem:
				P2		

38	d3.select(«...»).attr("width", «...»).attr("height", «...»)		Recognize? Yes purpose: It sets the width and height of an existing element. problem:	P2		
39	d3.«?1»(y)		Recognize? No purpose: ?? problem: too few information	NotP2	Abstract-Overall (P2)	

40	var svg = «...».append("g").attr("transform", "translate("
	+ «...» + "," + «...» + ")");
	d3.«?3»(«?1», function («?4»){
	}
	)
	d3.«?5»(«?1», function («?4»){
	}
	)
	«...»(«?1», function («?4»){
	}
	)
	«...».attr(«...», «...»)
	«...».attr(«...», «...»)
	var «?2» = «?6».selectAll(«...»).data(«?1»);
	«...».attr(«...», «...»).attr(«...», «...»)
			Recognize? Yes
			purpose:
			The first line creates an element "g" and moves it to specific locations
			problem:
			The code does unrelated things...
			[lines 2-10 are circled, saying "too abstract"]
				P2	Abstract-Fn (P2)
					Abstract-Call (P2)	