Suggesting API Usage to Novice Programmers with the Example Guru

Michelle Ichinco, Wint Hnin, and Caitlin Kelleher
Washington University in St. Louis
St. Louis, MO, USA
{michelle.ichinco, hnin, ckelleher}@wustl.edu

ABSTRACT

Programmers, especially novices, often have difficulty learning new APIs (Application Programming Interfaces). Existing research has not fully addressed novice programmers' unawareness of all available API methods. To help novices discover new and appropriate uses for API methods, we designed a system called the Example Guru. The Example Guru suggests context-relevant API methods based on each programmer's code. The suggestions provide contrasting examples to demonstrate how to use the API methods. To evaluate the effectiveness of the Example Guru, we ran a study comparing novice programmers' use of the Example Guru and documentation-inspired API information. We found that twice as many participants accessed the Example Guru suggestions compared to documentation and that participants used more than twice as many new API methods after accessing suggestions than documentation.

ACM Classification Keywords

H.5.2 [User Interfaces]: Evaluation/method

Author Keywords

APIs; novice programming; programming support; examples

INTRODUCTION

Research has shown that programmers often struggle to learn and use Application Programming Interfaces (APIs) [47]. These issues learning APIs stem from a variety of causes, including insufficient resources, confusing API structure, lack of programming experience, and unawareness of API methods [46]. This paper will focus primarily on API method unawareness. While unawareness of API methods affects all programmers, those with less programming experience, such as children learning programming or end-user programmers, often find barriers to learning APIs insurmountable [28].

We are unaware of existing research in API support or computer science education that has fully addressed the awareness problem in learning APIs. Instead, researchers have created systems for helping experienced programmers use APIs that improve: code completion [26], search [50], and available documentation [49]. These support systems require users to query a method of interest, so they do not help programmers identify new applicable API methods or incorrect usages of API methods.

To illustrate why using a large and unfamiliar API can be especially challenging for non-expert programmers, imagine an end-user programmer, Julie, who needs to analyze data from a biology study quickly. She decides to use Ruby to write a CSV file of results, but does not realize that an API method exists to automatically format an array correctly with commas [8]. Instead, she loops through her data, adding commas where they seem appropriate. Existing commas within her data make this task even more complex. Imagine instead that while Julie writes her array output code, her IDE offers a tip that introduces Julie to the method used for array formatting along with examples that illustrate its use.

In this paper, we introduce a system called the Example Guru, and evaluate its impact on API exploration and use. The Example Guru is designed to suggest relevant API information while programmers work on their own projects. The Example Guru uses a set of static rules to automatically check novice programs for opportunities to suggest API information. When it finds opportunities to do so, the Example Guru offers quick suggestions of relevant API methods. The user can expand the suggestions to view two contrasting executable code examples and support for finding and adding relevant code elements. To evaluate the effectiveness of the Example Guru at encouraging new API use by novice programmers, we ran a study comparing the Example Guru's suggestions to an in-application API documentation condition designed to capture the current best practice for supporting API use. Results show that twice as many novice programmers using the Example Guru accessed suggestions as accessed the in-application documentation in the control condition. Novice programmers also used over twice as many new API methods after accessing suggestions than after accessing documentation. Overall, we found that our suggestions helped broaden novice programmers' use of a new API. This paper has two contributions: 1) a system that motivates use of new API methods through context-relevant suggestions and contrasting examples, and 2) a study demonstrating increased exploration and use of API methods over best practice in-application documentation.

CHI 2017, May 06-11, 2017, Denver, CO, USA

 ISBN 978-1-4503-4655-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3025453.3025827

RELATED WORK

This section focuses on how related systems recommend support to programmers and users of complex software. Complex software systems are related because large numbers of available commands create problems similar to those that programmers face when learning new APIs. In the System Design section, we will discuss how the design of the interface and examples compare to similar systems. Here, we will place the Example Guru within the context of work on recommending support based on: 1) the behaviors of communities of users, 2) the ways individuals program and use complex software, and 3) the errors users encounter.

Recommending support based on communities of users

Existing systems support novices in learning APIs, programming, and complex software by leveraging: 1) overall community usage, and 2) community example repositories.

Overall command usage

Some API and software support uses community data to recommend commonly used commands. Systems provide recommendations with ranked lists based on common usages in a variety of ways, such as: lists of API methods in a programming environment [19], lists of commands within software [35], or by emphasizing more commonly used API methods in documentation [49]. Some recommendation tools use collaborative filtering algorithms, which classify a user's behavior within community usage data in order to recommend API methods [37, 44] or software commands [35, 30].

Recommending from sets of community examples

Research has also leveraged communities of users to find examples, Q&A information, and code completion methods related to a user's program. Systems work to improve example code retrieval for supporting programmers in using APIs. They do this by comparing users' code against repositories [20], mining patterns of APIs often used together [54], looking for related words to find code examples for similar types of functionality [2], or using input and output types [50, 33]. For more project-specific examples in open-source projects, recommendations have been based on the program history and types of tasks [10, 32]. Other than examples, systems also use community resources to inform relevant Q&A recommendation [9], code completion [5, 26], and parameter completion [1, 53].

The Example Guru also uses information about how a community of users commonly uses API methods, but it uses that information along with the context of the code. Because the Example Guru aims to improve the API unawareness issue, it suggests examples, rather than improving the ease of searching for examples. Furthermore, when the Example Guru suggests examples, the suggestions present new API methods that the programmer may want to learn, rather than examples about what the user is already doing.

Recommending support based on individual usage

Some systems provide recommendations for effective programming, APIs, or software commands based on either the user's behavior or the artifact they are working on.

Behavior

Research has used current or past behavior to recommend support to users of complex software and APIs. AmbientHelp recommends information based on the commands a user is working with at any point in time [34]. Similarly, CoDis suggests unfamiliar commands based on command patterns and the time since the user's last activity [55]. Another tool bases API recommendations on the user's programming history [45].

Artifacts

Some systems use just the artifact the user is creating to recommend help for APIs, programming, and commands in complex software. Tools for recommending APIs consider the structure of code to recommend API methods, such as by looking for redundant code [25] or using identifiers from a class's abstract syntax tree [18]. Documentation recommendations can also rely on artifacts, like by connecting method invocations to documentation [11], or by relating software interface elements to documentation [27]. Systems also recommend commands to users in sketching software based on the drawing artifacts that users create [14, 22, 40].

These systems are the most similar to the Example Guru because the Example Guru also makes suggestions based on the programmer's code. However, these systems primarily focus on recommending commands, while the Example Guru also focuses on the user's motivation to access the suggestions. To do this, the Example Guru suggests unknown or incorrectly used API usages in order to both introduce new API methods, as well as to improve the output of the programmer's code. Results-oriented programmers like novice and end-user programmers will likely be more motivated by suggestions that explicitly connect new API methods to the output of their code.

Recommending support based on errors

Software systems and programming environments also suggest information to users based on errors. Tools for non-expert programmers recommend information to try to help users who have hit a barrier in completing a code task [23] or who have errors in their code [17]. Two specific scenarios where recommendations based on errors can be especially useful are: in pasting and adapting code examples [12], and in complex software systems where commands are easily mistaken for each other [29]. While these systems effectively suggest examples to help resolve errors, they do not necessarily introduce new skills and require that the user has hit a problem in order to know what to suggest. Instead, the Example Guru suggests context-relevant API methods based on programmers' code in order to help them use new API methods or use API methods correctly.

THE EXAMPLE GURU

This section will first give an overview of how the Example Guru works, followed by the rationales for the interface design and the content the Example Guru suggests.

System Overview

The Example Guru suggests API usages to novice programmers based on their code in Looking Glass [31], a blocks-based programming environment for creating 3D animations designed for children aged 10-15 (see Figure 1). Looking Glass users are similar to end-user programmers in that they

are motivated by the output of their code, which in this case is an animation. The Looking Glass API is unique from many other common blocks-based APIs because it has methods such as 'walk,' 'resize,' and 'setTransparency' that perform operations on 3D graphics. We chose to implement the Example Guru within Looking Glass for three reasons: we wanted to address novice API use, Looking Glass users generally work on open-ended projects which was our target context, and we needed an unfamiliar API to introduce, so we could not use the more widely known blocks programming environments.

The Example Guru has two main features: 1) rules, which parse code, looking for opportunities to suggest API methods, and 2) suggestions, which include textual tips, contrasting code examples, and the 'show me' capability that demonstrates where to find an API method block in the interface. In essence, each rule asks a yes or no question about the presence of specific code elements in a program and triggers suggestions if the required code elements exist. For example, one rule checks for a character turning to face another character. The suggestion for that rule introduces joint movements, which allow a character to turn just their head to face something, rather than their whole body. The Example Guru uses rules to check programs each time a programmer executes their code. It then makes a suggestion to the user based on a triggered rule. The system will only make a suggestion for one rule at a time, so it uses an established priority if multiple suggestions arise at once. We provide more details about the suggestion priority in the system design section.

System design

In order to design the Example Guru we used two methods: 1) formative studies, and 2) program analysis. For the Example Guru interface and suggestions, we used an iterative design process in a formative study with 48 participants aged 10-15. To design the rules and suggestions, we used two sets of programs not created for this study. One set contained 107 programs created by Looking Glass API experts. The second set contained 600 programs shared to the Looking Glass website by non-experts [31].

Rules

For this study, we designed and implemented the rules by hand. In the discussion, we address how a system could automate this process in order for it to apply to other and larger APIs. Our process had three main steps: 1) compare novice and expert API use to select the API methods to suggest and the priority order, 2) consider the types of animations experts created with specific API methods and find simpler or related animations novices make where new API methods could be useful, and 3) author the rules within the system.

In order to select which API methods to suggest and the priority with which to suggest them, we compared our sets of novice and expert programs. We wanted to suggest API methods that novices were likely to be unfamiliar with, but also that they were likely to find useful. The set of API methods the Example Guru suggests contains API methods that experts used more often than novices and that experts used more than 5% of the time. These API methods are likely unfamiliar, but also used frequently enough by experts to be useful. Selecting the API methods to suggest based on expert usage helps to prevent the suggestions from over-fitting to the first hour of programming. Since the Example Guru only presents one suggestion at a time, we designed a priority ordering for selecting one of the triggered rules to suggest. Rules suggesting ways to correct API usages have the highest priority. The Example Guru then suggests API methods with the largest difference in expert and novice use and that experts used more often. The lowest priority suggestions are for API methods that experts and novices used with similar frequencies or that experts rarely used.

The design of rules is similar to code smells [13] and anti-patterns [23], but instead of focusing on checking for poorly composed code, most of the rules in the Example Guru look for opportunities to introduce new concepts. Essentially, rules recommend ways to improve animations through the use of previously unused API methods. Our formative work showed that it is important for the rules to find opportunities to improve novices' animations because suggestions that only improved the code quality were less exciting to novices creating animations. This is because novices in Looking Glass are focusing primarily on their animations, rather than on trying to learn new programming concepts. In order to decide when to recommend a particular unused API method, we manually checked how experts used API methods for complex animations. In many cases, novices create similar, more basic animations with more commonly used API methods. For instance, one rule checks programs for characters turning multiple rotations, as the programmer may be attempting to make characters dance. The rule has a suggestion that demonstrates how to animate joints to make a more realistic dancing animation. Each rule has an associated suggestion that introduces the API usage to the programmer.

Finally, we implemented the rules within the Example Guru. Rules contain a specification of how to parse code for opportunities to improve. Specifications use an internal API designed to simplify querying the abstract syntax tree.

Suggestions

We designed the suggestion presentation and examples through an iterative process with one-on-one study sessions.

We designed the suggestion presentation with the following goals: 1) to not interrupt or overwhelm the user, 2) to be easily accessible, and 3) to demonstrate the relevance of the suggestion to the code. Formative user testing indicated that programmers were most open to new ideas and improvements around the time they decide to test their code, so the Example Guru presents new suggestions after code execution. A list of suggestions allows the user to return to a suggestion at any point (see Figure 1-A), while the code annotations connect suggestions to the relevant code (see Figure 1-B). Hovering over a suggestion in the suggestion list provides a preview of the example and hovering over an annotation in the code shows a text description of the suggestion. These previews provide a hint of what the suggestion would show if opened, similar to surprise, explain, reward [51].

The examples presented within suggestions differ from those in other systems [17, 23, 41] in two critical ways: 1) they emphasize how the API method works using two contrasting

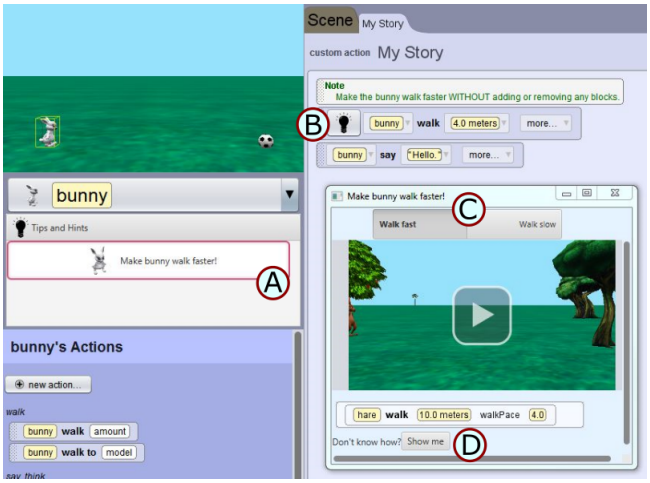

Figure 1. The Example Guru implemented within Looking Glass. (A) List of all suggestions. (B) Code annotation button to open the most recently added suggestion. (C) Contrasting examples such as 'walk fast' and 'walk slow'. (D) 'Show me' button that users can click to see the location of the suggested block.

examples (see Figure 1-C), and 2) they provide support for finding the relevant code block in the interface (see Figure 1-D). We developed the idea for contrasting examples through formative testing, where participants often did not know which argument values to use in blocks of code. The two contrasting examples either show different values or two API methods that work similarly in order to highlight the differences. The goal of the contrasting examples was to encourage novices to perform self-explanation, which has been shown to be effective for learning [7]. Within a blocks environment, understanding how a block works does not necessarily mean a novice can use it. Formative and related work indicated that novices sometimes have trouble finding a code block from an example [21]. One study [16] found that providing a 'show me' button that, when clicked, highlighted the location of the block in the programming environment, helped programmers find necessary blocks (see Figure 1-D). In the results, we answer whether participants used the features provided in our design. Table 1 details designs we tested and found to be ineffective.

EVALUATION

We ran a study to evaluate the effectiveness of the Example Guru's suggestions in encouraging new API method use by comparing them to an in-application documentation control condition. We will call the two conditions the 'suggestions' condition and the 'documentation' condition. In working towards reducing the unawareness problem for novice programmers learning new APIs, we tested the following two hypotheses:

H1: Novice programmers will access suggestions more frequently than documentation. We hypothesize that the suggestions will expose novice programmers to API methods that they likely would not have realized existed.

H2: Novice programmers using suggestions will improve their API usage more as a result of suggestions than novice programmers will from API documentation. Here, we want to compare the number and type of API methods participants add to their code after accessing suggestions or documentation.

Design	Issue
Suggestions appeared alongside the execution view.	Users did not focus on suggestions while executing their code, but instead returned to the code before considering what to do next.
Suggestions only appeared as buttons next to the code.	Users did not always want to access suggestions immediately, but displaying many suggestions crowded the editor.
Suggestions only contained one example.	Users did not understand the impact of argument values relative to their code.
Examples had text along with the code to explain how the example worked.	The text made the example view crowded and made it hard for users to focus on the critical elements. Users rarely read the text.

Table 1. Unsuccessful design attempts in formative testing

Documentation Condition

Currently, the best practice for supporting use of unfamiliar API methods is providing easy-to-access documentation containing example code. A few systems use suggestions within a programming context, but focus on violations of proper programming. Errors provide natural motivation to use suggestions, but in the case of API use, we cannot assume that novices will be motivated to apply a nonessential suggestion. We believe comparing to the best practice, documentation, is an appropriate first step toward evaluating the Example Guru.

We designed the documentation based on two common forms of API support: online API documentation and code completion. We wanted the documentation to have full information like online documentation, while making it easily accessible like code completion. Thus, the user can access a doc (documentation for a specific API method) by clicking a '?' button beside the code block the user is interested in (see Figure 2-A). Upon opening a doc, the user can view descriptions and examples of how the API method works, along with all of the available parameters for that API method (see Figure 2).

Participants

We recruited participants who had never used Looking Glass because this study investigates novice programmers exploration and use of the Looking Glass API. We recruited 81 participants aged 10 to 15 from a local STEM mailing list. Two participants had used Looking Glass in the past and a third skipped the first phase of the study, so we analyzed the data from the remaining 78 participants. The 78 participants had an average age of 11.8 (SD= 1.6), were 46.2% female, 52.6% male, and 1.2% unspecified gender. We compensated each participant with a $10 gift card to Amazon.com.

Methods

We created materials in order to measure API information access, API usage, and participant features that could influence how participants use API information.

API information access and usage

In order to evaluate whether participants accessed suggestions more than documentation, we needed to ensure two things:

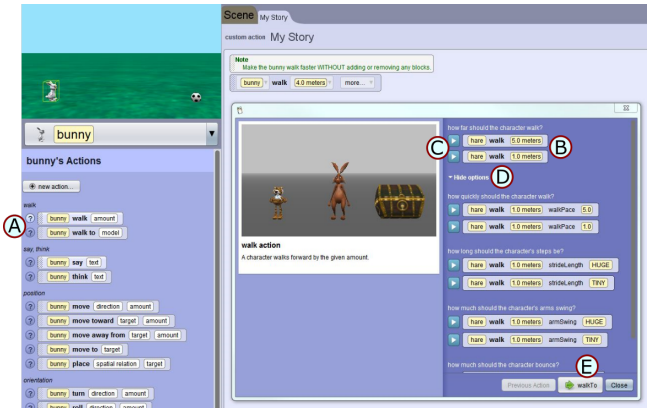

Figure 2. In-application API Documentation condition. (A) Users can access documentation using the '?' button available beside APIs. (B) Examples with different values and the description. (C) The play button can be used to execute the code. (D) Button to expand or collapse the parameters information. (E) Users can navigate to other doc using these buttons.

1) that participants were equally familiar with the API information formats, and 2) that participants actually received suggestions while working on open-ended programs. In order to familiarize participants with the API information, we created two training tasks. For the first training task, participants had to make a bunny walk faster by adding an optional argument value to a 'walk' action. For the second, they had to make a shark swim around an island by adding an optional argument value to a 'turn' code block. We provided instructions on a sheet of paper that directed participants to use the API information provided.

In order to improve the odds that participants would receive suggestions during open-ended programming, we created and tested scenes with props and characters for participants to use in creating their animations. For instance, complex movements and rotations trigger suggestions, so one scene was designed for a Seaworld show animation. This scene often motivated novice programmers to try to create complex animations with dolphins, which sometimes triggered suggestions. We selected five scenes for this study based on popular scenes from formative work because children were most excited to create animations with those scenes.

Participant features

In designing the Example Guru, we wanted novice programmers to benefit from suggestions regardless of their age, gender, or programming experience. Because the suggestions are context-relevant, we hypothesized that the suggestions would interest users with very little programming experience, as well as novices with more exposure. In order to capture information about programming experience, participants filled out a demographic and computing history survey.

We also thought that the way participants like to learn might affect how they use API documentation and explore new API methods. In order to capture this, participants also filled out an exploring and learning technology survey on paper before the study, modeled after the survey about trying new technology in [6]. Additionally, to better understand participants' motivations in using new API methods, we created dynamic surveys for participants to fill out on-screen after completing open-ended programs during the study. The surveys asked questions about why participants used new API methods for the first time and why they used or did not use API information during the just-completed program.

Study Procedures

There were three phases of this study: 1) baseline open-ended, 2) training, and 3) supported open-ended. The study was between subjects with two conditions: suggestions and documentation. Because work has demonstrated that gender plays into exploration and learning in software [6], we randomly assigned participants to either use suggestions or documentation keeping gender balanced. Participants worked individually on all tasks and were allowed to move onto the next task if they felt they had completed the current one.

Baseline open-ended phase

We wanted to know how participants would use the API without any support, so participants first created an animation without API support for up to 15 minutes. Because some participants had no programming experience, the instructions for the first phase gave information about how to drag blocks into animations and execute the animations. This phase involved open-ended programming, which means that there was no correct or incorrect answer and that participants were allowed to freely create their animations. We assigned participants a specific scene for this task and balanced the assignments of scenes across participants to limit any effect of specific scenes on API usage. In order to find out more about why participants added new API methods, participants completed an on-screen survey at the end of this task that asked about: 1) one new API method participants added and executed, and 2) one that they added, executed and deleted, if these existed. Due to space constraints, we will not discuss this survey in this paper.

Training phase

Due to time constraints for a controlled study, we wanted participants to become quickly comfortable with using either suggestions or documentation. To do this, we had all participants complete two training tasks. In both cases, the instructions showed how to access the suggestion or documentation that would help them complete the task. The researchers checked the participants' code to make sure they successfully completed the task and helped participants if needed. If participants completed a task without a suggestion or documentation, the researcher demonstrated how they could have used it to ensure that all participants were exposed to suggestions or documentation.

Supported open-ended phase

Finally, we wanted to evaluate how participants used the API information and API methods when working on their own projects. During the supported open-ended phase, participants created open-ended animations with either suggestions or documentation available to them. We first asked participants to create a program based on the idea of a Seaworld show. The purpose of providing an idea was to give participants a goal to work towards, but not to constrain what code they should use. Next, participants were assigned a scene in which they could create any animation or use a provided story prompt if they did not have an idea. Participants had up to fifteen minutes to work on each of the two animations. If participants finished

early, they could select a scene they had not yet used and create another animation. At the end of each of these animations, participants also typed answers to questions onscreen about why they added or removed certain API methods and why they accessed or did not access API information.

Data Collection & Analysis

We logged all actions participants took and survey answers to analyze suggestion, documentation, and API usage.

Time on task

We did not require participants to spend the full amount of time provided on each task, so some participants spent less than the standard amount of time. Most participants (76%) spent the full amount of time on the baseline (15 min.) and supported open-ended (30 min.) phases. We stopped analyzing participants' data after 30 minutes in supported open-ended. We will report the results for the set of participants who spent the full amount of time (59 participants), as well as the results for all participants.

Accessing API information and API usage

We analyzed logs in order to measure which suggestions and documentation participants accessed, meaning that they clicked to open the API information. To determine whether participants used new methods from the API information in their programs, we measured which API methods participants inserted into their programs for the first time after accessing related API information. When comparing the number of accesses and API usage, we used t-tests to compare the aggregate numbers because participants received different numbers of suggestions. Additionally, participants in the API documentation condition could access docs many more times than the number of suggestions available. We use Cohen's d to measure effect size (small: .3, medium: .5, large: .8). We also report the percentages of participants who accessed API information and used API methods and compare this using Chi-squared tests. We use the odds ratio to measure effect size (small: 1.5, medium: 3.5, large: 9).

For both API information access and API usage, we describe the kinds of API methods participants were accessing information for and inserting into their programs. We believe the best way to do this is to group the API methods based on how much novice programmers generally use them. We base the frequency of novice use on the set of 600 non-expert programs described earlier. We will discuss the API methods in terms of 4 groups: those that the Example Guru did not suggest, APIs suggested that were used least frequently by novices (the bottom third of usage frequency), those suggested that were sometimes used (the middle third), and those suggested that were most often used by novices (the top third of API method usage frequency).

Participant qualities

We analyzed participant qualities to try to understand the types of novice programmers who will benefit from suggestions or documentation. We collected gender, age, programming experience, and learning style data from the surveys. We captured programming experience using two survey questions: 'Have you programmed before?' and 'Have you programmed for more than 3 hours in your whole life?' Those who had less than 3 hours of programming experience likely only programmed once or twice without much instruction or practice. Nine participants in the suggestion condition (23%) and eight participants in the documentation condition (21%) had 0-3 hours of programming experience. We also intended to capture personal preferences about using API documentation using the on-screen end-of-task surveys for both conditions. Due to a technical error, the survey questions asking participants about why they did or did not access documentation did not appear for the study participants, so we report quotes from pilot users who completed the same study and received these questions.

RESULTS

We hypothesized that participants who received suggestions would: 1) access suggestions, and 2) use API methods from the suggestions more frequently than participants would access and use in-application documentation. In this section, we first explore these two hypotheses and then delve into how different participants used the API information and the features they used.

Access and use of suggestions and documentation

Ideally, suggestions should encourage API exploration when novice programmers are pursuing their own projects. We evaluated this through the number of times participants accessed API information and how many new API methods they used after accessing API information.

Accessing suggestions and documentation

We found that more participants accessed suggestions than accessed documentation: 82% of suggestion participants and 41% of documentation participants accessed at least one entry. The difference in the number of participants who accessed suggestions verses documentation was significant with a medium effect size ($\chi^2(1) = 12.19, p < 0.001$, odds ratio = 6.4). Participants in both conditions described using suggestions and documentation to gain additional information about API methods that seemed potentially relevant. A participant in the documentation condition described opening an API method that changed a character's appearance because: "... I wondered what it was. It turned out to change Alice." One participant in the suggestion condition sought additional information about a new method based on the tip offered as part of the suggestion: "I opened the tip for 'setTransparency' because I thought it was good way to make an object disappear".

We found that participants accessed more total suggestions, on average, than documentation. For all 78 participants, participants accessed significantly more suggestions (M=3.3, SD=2.7) than documentation (M=1.4, SD=2.7), $t(76) = 31, p < 0.01$, $d = 0.69$. Since some participants spent less than the full task time, we also confirmed that this difference existed for the set of participants who used the whole task time. The results were very similar: participants accessed suggestions (M=3.0, SD= 2.7) significantly more than documentation (M= 1.1, SD = 1.7) with a large effect size ($t(50) = 3.3, p < 0.01, d = 0.85$). Simply accessing more suggestions is a potential benefit to novice programmers because the suggestions expose them to broader range of API methods that may be useful either immediately or in the future. In this study, we could not measure whether a participant used a suggestion based on reading the tip without opening it, but survey

responses indicate that some participants did this: Participant S33 did not need to access a suggestion because reading it was enough: "I did not open the tip for Turn to Face because I read the outline for the Tip and used it in my code." Similarly, participant S70 said: "I did not open the tip because I saw it from the outside and felt like I could figure it out and I think I did."

Our goal was to encourage novice programmers to use API methods they would not necessarily use on their own. To evaluate this, we analyzed the information access and API use based on how often novice programmers in our sample set of programs used API methods. We split the API use based on one group of API methods that the Example Guru did not suggest and three groups that the Example Guru did suggest: the top third of methods that novices generally use most frequently, the middle third, the bottom third. The set that was not suggested includes API methods that novices use more than experts or that experts use in less than 5% of programs. In all three groups of API methods, the API information was accessed and used more frequently by participants in the suggestion condition (see Figure 3). While the largest use of suggestions was for API methods novices generally use the most, increasing use is beneficial because the average percentage of novices using those API methods is less than 50%. Furthermore, only participants with suggestions accessed information for the least used API methods.

The survey results provide additional insight into the reasons participants chose to access or not access API content. Due to a technical failure, participants in the documentation condition did not receive questions about their documentation access or use. Since questions about usage might encourage some users to increase their API usage, we looked for an increase in suggestion access and usage following the survey, which was administered after the first supported open-ended animation. However, we see little evidence of this. Ten participants used suggestions only during the first open-ended animation, an additional seventeen accessed suggestions throughout, and only five participants accessed zero suggestions during the first open-ended task, but one or more in the following animations. Thus, we do not believe that the survey questions influenced suggestion use.

Overall, participants described accessing suggestions to gain additional information about API methods that seemed potentially relevant. For example, one participant received a suggestion about setting the color of the sky, which they thought they could use: "The dark sky was sooo boring, so I looked at the tip and used it." Overall, participants opened 33% of the suggestions offered. For some participants, decisions not to open suggestions indicated lack of interest in those suggestions. In other cases, participants wanted to open all of the suggestions, but had not yet done so, like one participant: "I didn't open all of the tips yet." Finally, some participants were focused on other suggestions and missed ones that would have been of interest. One participant described missing a suggestion "because I was looking at other tips and didn't realize there was a tip [to] make only alien's head turn."

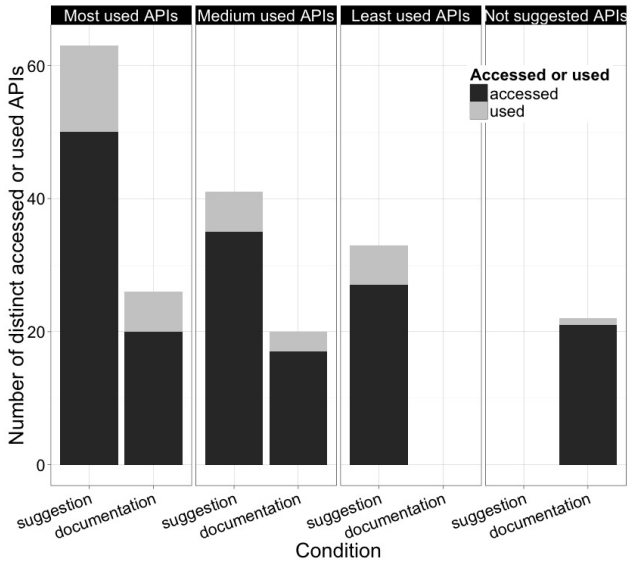

Figure 3. API information accessed and used grouped by frequency of API use by novice programmers.

Participants in the documentation condition similarly described a desire for additional information as a motivation for opening documentation: "I wanted to know what it was and I used it because I thought it would be pretty cool to begin and end abruptly." We unfortunately cannot report on their decisions around documentation they did not access.

Using suggestions and documentation

Since increased access to API information may help to support the use of a new API method, we also wanted to explore the use of new API methods after information access. We found that more participants used new API methods after accessing suggestions than after accessing documentation. About three times as many participants in the suggestions condition used an API method after accessing the API information as in the documentation condition, 38% vs. 12.8% ($\chi^2(1) = 5.4466, p < 0.05$, odds ratio = 4.17). Additionally, participants added more new API methods after accessing the suggestions (M= .59, SD=.82) than after accessing the documentation (M= 0.15, SD = 0.43). This was significant for all 78 participants ($t(57.6) = 2.94 p < 0.01, d = 0.67$) and for the 59 participants who used the full task time ($t(49.3) = 2.2, p < 0.05, d = 0.55$).

In addition to frequency of use, it is interesting to explore the diversity of methods participants choose to use. In particular, we designed our rules and suggestions with the goal of introducing API methods that experts use more frequently than novices. Participants in both conditions used more new API methods from the group of API methods that are most commonly used by novices than the other groups. However, we note that participants using suggestions used more new methods from the middle and low use categories combined (see Figure 3). Finally, we looked at API methods for which we did not create suggestions. While some participants in the documentation group accessed information about these methods, very few were actually added. This provides some support for our method of selecting API methods for suggestions.

Our survey results suggest that participants in the suggestions condition decided to use an API method based on its potential to improve their animation. One participant explained "I just thought that changing the posture of the dolphins created a more natural feel than just moving its entire body." In contrast, participants in the documentation condition more often cited goals of understanding. For example, one participant using documentation stated, "I opened it and chose to use it so I could see what it looked like." We see a similar dichotomy around participants' explanations for non-use. A participant in the suggestions condition chose not to use an accessed suggestion because it did not mesh with her vision for her story: "I wanted to have the dolphin to go different distances showing they each do a little more than the last dolphin." A participant in the documentation condition explained accessing but not using documentation for a duration parameter because "...I wanted to see how it worked."

Do participants' demographics affect how they used suggestions and documentation?

In the design of the Example Guru, we hoped to support participants regardless of age, programming experience, and gender. By having suggestions relate to the context of the program and API methods that the programmer had not yet used, we hypothesized that the suggestions should continue to be relevant to programmers as they become familiar with more of the API. Previous studies have found a correlation between age and programming success with the same age range of children [15]. These differences in performance could result from the developmental changes that impact children's abilities to understand abstraction around the ages of 11-12 [42]. We hoped that the context-relevant approach would support novice programmers of differing ages. We also hypothesized that suggestions might better support participants who liked to learn by accessing information, rather than by tinkering, since suggestions do not require the user to seek out new API methods and documentation. Since females have been shown to be less likely to learn through exploration in some cases [3], it seemed as though the suggestions might provide better support for female novice programmers.

Age and Programming Experience

Our results did not show a relationship between age and accessing and using either suggestions or documentation. Specifically, we found no significant correlation between age and suggestion access($t(37) = -0.5, p = 0.62, r = -0.08$) nor between age and documentation access ($t(37) = 0.37, p = 0.71, r = 0.06$). Similarly, we found no significant correlation between age and the number of API methods used after accessing suggestions ($t(37) = -0.92, p = 0.36, r = -0.15$), nor between age and the number of API methods used after accessing documentation ($t(37) = -0.22, p = 0.83, r = -0.04$). These results suggest that both documentation and suggestions are used similarly by children ranging in age from ten to fifteen.

Programming experience played a larger role in how much participants accessed and used API information. Those with less than three hours of programming experience were the most likely to access both suggestions (100%) and documentation (75%). Participants with little programming in both conditions added new API methods after accessing them at similar rates: 44% of those in the suggestions condition and 38% of those in the documentation condition added API methods. However, among those with more than three hours of programming experience, we see a trend towards more use of suggestions. Of the participants in the suggestions condition, 77% accessed a suggestion and 40% added a new API method after accessing its suggestion. For participants in the documentation condition, only 32% accessed API information and 6.5% used a new API method after accessing its documentation. This trend suggests significant promise in the use of context-relevant API suggestions to help programmers continue to explore new API methods.

Gender and Learning Style

When considering how gender might relate to participants' use of API information, we explored use by reported gender, as well as learning style based on a survey.

We found that both males and females accessed and used the suggestions at higher rates than the documentation. However, male participants accessed suggestions more often than female participants: males accessed an average of 4.3 suggestions as compared to 2.2 suggestions for females ($t(33.5) = -2.7, p < 0.05, d = .83$). Male participants also accessed a larger percentage of the suggestions they received, so the larger number of suggestions accessed by males was not due to a larger number of suggestions received ($t(34.3) = -2.7, p < 0.05, d = 0.84$). While not significant, we note that female participants accessed documentation more often than male participants, averaging 1.9 documentation accesses as compared to 1.05 for males, as shown in Table 2. Overall, male participants opened more suggestions, but both genders accessed suggestions. The significant difference in terms of the number of suggestions accessed represents an important avenue for additional research. While suggestions performed better than documentation for both male and female participants, the lower usage by female participants has the potential to create an educational inequity.

One of the main personality traits that often correlates with gender differences in programming is the programmer's learning style: whether they like to learn by tinkering and exploring or using a step-by-step approach, so we also wanted to compare the way participants desired to learn and their behaviors. We created a survey based on the survey in [6] in order to try to determine whether participants were more likely to explore and tinker as a way of exploring the API or whether they were more reliant on information like tutorials or books. Unfortunately, the survey only had a reliability of $\alpha = 0.65$ for the questions about learning through exploring, and $\alpha = 0.5$ for the questions about learning using process-oriented information, both of which are less than the accepted reliability for surveys (0.7), so we will not report results for the survey.

Do participants take advantage of API information features?

This section presents results on how participants used features in the suggestions and documentation. Due to the structure of this study, we cannot evaluate the impact of specific features, so instead we explore three questions about feature use to provide insight into the value of the system design: 1) how did participants access information, 2) how much did they execute

Condition	Action	<3 hours prog.	3+ hours prog.	p	Male	Female	p
Suggestions	Accessed	100%	77%		86%	78%	
		M=3.8 SD=2.3	M=3.2 SD=2.9		M=4.3 SD=3.0	M=2.2 SD=1.8	<.05
	Added API call	44%	40%		48%	33 %	
		M=.56 SD=.72	M=.6 SD=.86		M=.67 SD=.86	M=.5 SD=.79	
Documentation	Accessed	75%	32%	.07	30%	56%	
		M=3.4 SD=4.0	M=.94 SD =2.0		M=1.05 SD=2.8	M=1.9 SD=2.7	
	Added API call	38%	6.5%	.08	10%	17%	
		M=.5 SD=.76	M=.06 SD=.25		M=0.1 SD=.31	M=0.22 SD=.5	

Table 2. Participant characteristics and information access and API usage.

examples, and 3) how much do they use contrasting examples and the 'show me' button?

We expected participants to access the suggestions and documentation using all of the mechanisms provided, which we found to be true, as shown in Table 3. For the most part, participants accessed suggestions from the suggestion list (see Figure 1-A) and '?' buttons (see Figure 2-A), which were both in or near the palette where users drag code blocks from. The list of suggestions was designed to help participants return to suggestions, which participants did: "I opened the tip [suggestion] because I had forgotten how to do it."

We found that the majority of participants who accessed API information also executed examples in both conditions, but did so more with suggestions: 81.3% of participants who accessed suggestions executed an example at least once, while 68.8 % of participants who accessed documentation executed at least one example. Executing examples may suggest that participants wanted to engage more deeply with the information in order to find out more about it. Participants who executed examples from suggestions executed on average 4.7 examples (SD = 3.5), while participants executed examples an average of 9.7 times from documentation (SD= 9). This may be because suggestions only provided two examples, while documentation often showed eight examples.

Because we designed suggestions specifically to provide contrasting examples and a button to help novices find code blocks, we measured how much they used those features. 44% of participants who accessed suggestions used contrasting examples and accessed the contrasting example for 1.8 suggestions on average (SD= 1.2). 38% of participants who accessed suggestions used the 'show me' button, and on average, clicked it 3.2 times (SD= 2.5). Since participants likely will not need these features for every suggestion, having over 30% usage and having participants return to use these features multiple times seems to indicate that participants found the features useful.

Condition	Way of accessing	% of accesses
Suggestions	Suggestion Panel	86.5%
	Code Annotation	11.1%
	Preview from Panel	2.4%
Documentation	'?' Button	45.1%
	Expanding Parameters	29.4%
	'More Examples'	17.6%
	Next/Previous Buttons	7.8%

Table 3. Participants accessed the API information all of the different ways in both conditions

DISCUSSION AND FUTURE WORK

Given that programmers across a broad range of skill sets describe learning or attempting to learn using 'just-in-time' strategies, effective situated support for API learning has the potential to improve programmer success and efficiency, particularly for novices. The results of our study suggest that the Example Guru approach has the potential to better support learning of APIs during open-ended programming. Yet, there are places where further work is needed. First, our results found that females used fewer suggestions than males, leading to a potential learning inequity. Second, we hand-coded our rules for this study. To enable approaches like the Example Guru to be used more broadly will require readily scalable approaches to generating effective rules.

Learning APIs

To achieve mastery of a new API, novice programmers must continue to develop their skills over time. Yet, existing research suggests that novices reach a plateau in which they quickly learn to use a subset of the available capabilities within the system and then stop learning new skills [52, 48]. One recent paper [36] found an increase in the number of API methods used with experience. However, the increase was small after the initial period. Although measures of API method use cannot tell us whether the novice programmers actually have a full understanding of how the API methods work, programmers must first gain exposure and experience with the API methods. Thus, this work begins to address issues in API learning by improving the number of API methods that novice programmers explore and use.

We believe that some of the plateau effect may be due to a lack of appropriate learning mechanisms. While users may consult tutorials and similar materials when getting started with a new system or API, the use of them tends to drop off in favor of open-ended programming and just-in-time learning [4]. Yet, just-in-time approaches to learning often require that programmers know the right method to query for information. The Example Guru approach shows promise in introducing novice programmers to a broader selection of API methods as they work on their own projects. Rather than requiring that they know what methods to search for, the Example Guru observes their code and offers potentially relevant information. Participants accessed suggestions and used API methods from suggestions more frequently than documentation, creating more potential learning opportunities. It is important to

note that participants accessed suggestions for API methods novices rarely use in common practice, while users in the documentation condition chose to explore mostly commonly used methods. Over a longer period, the increased exposure to and use of API information could lead to substantial learning gains.

Finally, our results suggest the potential for continued usage by those with varying skill levels. In both the documentation and suggestion conditions, participants with fewer than three hours of programming experience accessed and used API information more frequently than those with more than three hours of experience. The difference is much more dramatic in the documentation condition where only 32% of participants with more programming experience accessed the API information at all. In contrast, 77% of those with more than three hours of programming experience accessed the suggestions. Yet, there is still room for improvement. While 40% of the more experienced novices in our sample engaged with API methods from suggestions, 60% did not.

Gender and the Example Guru

Our results showed different usage patterns among male and female participants. Specifically, male participants accessed more suggestions than female participants, averaging 4.3 versus 2.2 suggestions accessed. This is a potentially troubling difference, as over time this can lead to an educational inequity. Based on the results of this study, we have little information about the reasons for this difference. Previous work suggests that females may prefer learning using step-by-step instructions, rather than through tinkering and exploring [24] and that females have a tendency toward comprehensive information processing versus males' tendency toward selective information processing [38, 39]. However, we note that this difference occurs based solely on the tip describing the suggestion and before users are in a position to do much information processing. This is an area where future work is needed in order to understand and address this difference.

Rules and suggestions at scale

The manual approach that we took to create rules and suggestions is a labor intensive one, so a natural direction for future work is to explore ways to partially or fully automate this process. Generalizing this work requires the ability to perform two processes at a larger scale: 1) generating the rules, and 2) creating interactive examples.

We authored our rules by hand, which involved manually considering how expert API usage could improve novice programs. We could reduce the amount of human effort necessary through automatic search of program corpora to identify patterns of API use. By incorporating positive and negative labels from resources like Stack Overflow, a system could likely make a reasonable decision about the correct ways experts use APIs and when an API method might be useful. However, some human intuition will likely be necessary to curate the final rules.

We designed our example code based on expert usages of API methods. A system could select example code automatically from online forum code with labels or a large corpus of expert code. We also need ways to automatically select contrasting examples and enable users to visualize examples. Contrasting examples could be automatically generated in some cases by changing parameter values. In other cases, API methods mentioned in forum threads may be relevant as contrasting API methods. This might be another area where a small amount of expert human opinion may be necessary, but a system could simplify this process by presenting a list of possible contrasting examples to let an expert quickly choose. Most example code, when run, will not automatically provide a visualization to help novices understand how it works. Approaches like those used in Python Tutor [43] and algorithm animation could provide visualizations of non-visual code.

Limitations

There are two limitations to this study: the population we picked and the length of the study.

We recruited participants from a mailing list focused on STEM which draws from a sample of more interested and self-motivated learners than the general population. This may have meant that participants were more interested in technology and excited to explore the API than the norm. Furthermore, 94% of participants had programming experience of some form, including 82% who had been taught programming, which is above the norm for middle school children in the US.

While the results from this initial study are exciting, it is important to note that this study focused on a relatively short period of time and on API use, rather than learning. While the patterns of use suggest the potential for improved longer term learning, it will be important to explore how novice programmers engage with the Example Guru over a longer period. We need further studies to understand whether the Example Guru improves novices' comprehension of the API methods.

CONCLUSION

Programmers at all levels prefer to learn while working on their own projects, but most existing tools provide API help using method names. We introduce the Example Guru, a system that suggests context-relevant API methods to programmers while they work on open-ended projects. Our study comparing suggestions and in-application documentation found that participants in the suggestion condition accessed and used information about the API methods more often than participants in the documentation condition. Further, the suggestions introduced participants to a greater diversity of methods. API learning is a fundamental part of programming activity today. Over time, this pattern of suggestion use could result in more efficient and higher quality use of unfamiliar APIs.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. 1054587 & 1440996. We would like to thank Kyle Harms, Dennis Cosgrove, and Randall Brachman for their feedback on this paper.

REFERENCES

1. Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. 2015. PARC: Recommending API methods parameters. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 330–332.

2. Sushil K Bajracharya, Joel Ossher, and Cristina V Lopes. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 157–166.
3. Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. 2006. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 231–240.
4. Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.
5. Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. 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*. ACM, 213–222.
6. Margaret Burnett, Scott D. Fleming, Shamsi Iqbal, Gina Venolia, Vidya Rajaram, Umer Farooq, Valentina Grigoreanu, and Mary Czerwinski. 2010. Gender differences and programming environments: across programming populations. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*. ACM, 28.
7. Michelene TH Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. 1989. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science* 13, 2 (1989), 145–182.
8. Class : CSV (Ruby 1.9.3,) 2016. http://ruby-doc.org/stdlib-1.9.3/libdoc/csv/rdoc/CSV.html. (2016). Accessed: 2016-09-20.
9. Joel Cordeiro, Bruno Antunes, and Paulo Gomes. 2012. Context-based recommendation to support problem solving in software development. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 85–89.
10. Davor Cubranic and Gail C Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 408–418.
11. Uri Dekel and James D Herbsleb. 2009. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 320–330.
12. Christian Dörner, Andrew R Faulring, and Brad A Myers. 2014. EUKLAS: Supporting copy-and-paste strategies for integrating example code. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 13–20.
13. Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
14. C Ailie Fraser, Mira Dontcheva, Holger Winnemöller, Sheryl Ehrlich, and Scott Klemmer. 2016. DiscoverySpace: Suggesting Actions in Complex Software. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*. ACM, 1221–1232.
15. Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. 2016. Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 241–250.
16. Kyle J. Harms, Dennis Cosgrove, Shannon Gray, and Caitlin Kelleher. 2013. Automatically generating tutorials to enable middle school children to learn programming independently. In *Proceedings of the 12th International Conference on Interaction Design and Children*. ACM, 11–19.
17. Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.
18. Lars Heinemann, Veronika Bauer, Markus Herrmannsdoerfer, and Benjamin Hummel. 2012. Identifier-based context-dependent API method recommendation. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 31–40.
19. Reid Holmes and Robert J Walker. 2008. A newbie's guide to Eclipse APIs. In *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 149–152.
20. Reid Holmes, Robert J Walker, and Gail C Murphy. 2006. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering* 32, 12 (2006), 952–970.
21. Michelle Ichinco and Caitlin Kelleher. 2015. Exploring novice programmer example use. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 63–71.
22. Takeo Igarashi and John F Hughes. 2001. A suggestive interface for 3D drawing. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*. ACM, 173–181.
23. Will Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Andrew Ko. 2015. A principled evaluation for a principled Idea Garden. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 235–243.
24. M. Gail Jones, Laura Brader-Araje, Lisa Wilson Carboni, Glenda Carter, Melissa J. Rua, Eric Banilower, and Holly Hatch. 2000. Tool time: Gender and students' use of

tools, control, and authority. *Journal of Research in Science Teaching* 37, 8 (2000), 760–783.

25. David Kawrykow and Martin P Robillard. 2009. Improving API usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 111–122.
26. Muhammad Asaduzzaman Chanchal K Roy Kevin and A Schneider Daqing Hou. 2014. CSCC: simple, efficient, context sensitive code completion. (2014).
27. Md Adnan Alam Khan, Volodymyr Dziubak, and Andrea Bunt. 2015. Exploring personalized command recommendations based on information found in Web documentation. In *Proceedings of the 20th International Conference on Intelligent User Interfaces*. ACM, 225–235.
28. Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.
29. Benjamin Lafreniere, Parmit K Chilana, Adam Fourney, and Michael A Terry. 2015. These Aren't the Commands You're Looking For: Addressing False Feedforward in Feature-Rich Software. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 619–628.
30. Wei Li, Justin Matejka, Tovi Grossman, Joseph A. Konstan, and George Fitzmaurice. 2011. Design and evaluation of a command recommendation system for software applications. *ACM Transactions on Computer-Human Interaction (TOCHI)* 18, 2 (2011), 6.
31. Looking Glass Community 2016. https://lookingglass.wustl.edu/. (2016). Accessed: 2013-02-24.
32. Yuri Malheiros, Alan Moraes, Cleyton Trindade, and Silvio Meira. 2012. A source code recommender system to support newcomers. In *2012 IEEE 36th Annual Computer Software and Applications Conference*. IEEE, 19–24.
33. David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 48–61.
34. Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2011. Ambient help. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2751–2760.
35. Justin Matejka, Wei Li, Tovi Grossman, and George Fitzmaurice. 2009. CommunityCommands: command recommendations for software applications. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 193–202.
36. J. Nathan Matias, Sayamindu Dasgupta, and Benjamin Mako Hill. 2016. Skill Progression in Scratch Revisited. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 1486–1490.
37. Frank McCarey, Mel O Cinneide, and Nicholas Kushmerick. 2006. A recommender agent for software libraries: An evaluation of memory-based and model-based collaborative filtering. In *Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*. IEEE Computer Society, 154–162.
38. Joan Meyers-Levy. 1986. *Gender differences in information processing: A selectivity interpretation*. Ph.D. Dissertation. Northwestern University.
39. Joan Meyers-Levy and Durairaj Maheswaran. 1991. Exploring differences in males' and females' processing strategies. *Journal of Consumer Research* 18, 1 (1991), 63–70.
40. Sundar Murugappan, Subramani Sellamani, and Karthik Ramani. 2009. Towards beautification of freehand sketches using suggestions. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*. ACM, 69–76.
41. Stephen Oney and Joel Brandt. 2012. Codelets: linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2697–2706.
42. Jean Piaget. 1972. Intellectual evolution from adolescence to adulthood. *Human development* 15, 1 (1972), 1–12.
43. Python Tutor 2017. Python Tutor - Visualize Python, Java, JavaScript, TypeScript, Ruby, C, and C++ code execution. http://pythontutor.com/. (2017).
44. Luisa Fernanda Hernández Ramírez and others. 2016. API recommendation system in Software Engineering. (2016).
45. Romain Robbes and Michele Lanza. 2008. How program history can improve code completion. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 317–326.
46. Martin P. Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.
47. Martin P. Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.
48. Christopher Scaffidi and Christopher Chambers. 2012. Skill progression demonstrated by users in the Scratch animation environment. *International Journal of Human-Computer Interaction* 28, 6 (2012), 383–398.
49. Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A Myers. 2009. Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 119–126.
50. Suresh Thummalapenta and Tao Xie. 2007. ParseWEB: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 204–213.

51. Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. 2003. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 305–312.
52. Benjamin Xie and Hal Abelson. 2016. Skill Progression in MIT App Inventor. *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on* (2016).
53. Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. 2012. Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 826–836.
54. Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.
55. Sedigheh Zolaktaf and Gail C Murphy. 2015. What to learn next: recommending commands in a feature-rich environment. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 1038–1044.