Towards a Model of API Learning

Caitlin Kelleher
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
ckelleher@wustl.edu

Michelle Ichinco
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA, USA
michelle_icinchco@uml.edu

Abstract—In today’s world, learning new APIs (Application Programming Interfaces) is fundamental to being a programmer. Prior research suggests that programmers learn on-the-fly while they work on other project-related tasks. Yet, this process is often inefficient. This inefficiency has inspired research seeking to understand and improve API learnability. While the existing research has provided insight into API learning, we still have a fractured understanding of the process of learning a new API. In this paper, we take the first steps towards developing a theoretical model of API learning by combining predictions from Information Foraging Theory (IFT) to describe information search behavior, Cognitive Load Theory (CLT) to describe learning, and External Memory (EM) to describe how API learners augment their short term memories. Our proposed model is consistent with existing research on barriers to learning APIs and helps to provide explanations for these barriers as well as suggest new research directions.

Index Terms—API Learning, Information Foraging, Cognitive Load Theory, External Memory

I. INTRODUCTION

Software plays an important and growing role in the US economy. In fact, the US Bureau of Labor Statistics predicts that the number of software developers will grow by more than 250,000 between 2016 and 2026, an increase of more than 30% [1]. Unlike many fields, software development is unlikely to be readily automatable, increasing the importance of improving programmer productivity. Application Programming Interfaces (APIs) decrease the amount of work necessary to build a new system by enabling reuse of code that others have already written and tested. API use is widespread. In fact, some estimate that if we include both APIs produced in-house for a project as well as those produced by external organizations, that almost all of the lines of code programmers write involve an API [44].

In a world of pervasive API use, learning new APIs is fundamental to being a programmer. The ProgrammableWeb, which lists only web-focused APIs, adds approximately 40 new web-based APIs to its database each week and contained over 19,000 APIs as of January 2018 [2], [65]. When starting a new project, programmers often need to learn one or more new APIs. Prior research suggests that programmers learn on-the-fly while they work on other project-related tasks [8]. During this process, programmers often struggle 1) to frame questions that address their information need [14], [64] and 2) to integrate multiple API elements [14], [60], [61].

The inefficiency of on-the-fly API learning [60], [61] has inspired a diverse set of research agendas with the end goal of improving API learnability including: programmers’ perceived API learning barriers [14], [60], [61], [64], how to design more usable APIs [10], [39], [51], [70], [71], how to improve API documentation [4], [15], [22], [30], [40], [46], [49], [62], [68], [71]–[73], and how to design tools that support API learning [5], [11], [19], [20], [47], [55]–[57], [66], [69], [77]. However, despite the diversity of API learnability topics covered in the literature, we still have a fractured understanding of the process of learning a new API.

In this paper, we take the first steps towards developing a theoretical model of API learning by combining predictions from Information Foraging Theory (IFT) to describe information search behavior, Cognitive Load Theory (CLT) to describe learning, and External Memory (EM) to describe how API learners augment their short term memories. This proposed model creates a new lens through which we can analyze API learning behavior in the context of existing tools and identify opportunities for further study and new support tools.

II. THEORETICAL BACKGROUND

We propose a theoretical model that describes task completion using an unfamiliar API grounded in three areas of research: Information Foraging Theory, Cognitive Load Theory and External Memory.

A. Information Foraging Theory

When completing a task using an unfamiliar API, a programmer must search for and process information related to that task. Information Foraging Theory (IFT) provides a predictive model of search behavior based on animal foraging behavior [53]. IFT predicts that users will attempt to maximize the ratio of the value of found information to the cost of obtaining that information. In each information patch, or document, users make a decision about whether to process information in that patch, navigate to a connected patch, or enrich the information environment [53]. Examples of enrichment include decreasing the cost of re-finding information (e.g. creating a bookmark) or generating a new information patch using a search engine [16]. We are not aware of anyone having applied IFT within the domain of API Learning. A number of researchers have identified information foraging behavior among programmers [6]–[8], [12], [13], [24], [63], applied IFT to programming-related domains including debugging [16],
B. Cognitive Load Theory

Cognitive Load Theory (CLT) observes that working memory is a bottleneck in learning tasks [18] and describes three types of working memory loads that can impact learning: intrinsic load, extraneous load, and germane load [50], [75]. A learning task’s intrinsic load is determined by the task’s nature and the learner’s expertise and is generally considered unalterable [75]. Most adults can easily sum small integers, for example, but a young child may struggle. Extraneous load consists of tasks that are not directly related to the specific learning goals [75]. This load can be imposed by activities like unnecessary searches for information [23] or a need to integrate different sources of information [74], [75]. Germane load represents extra effort that can support learning. Activities such as identifying and explaining a problem’s sub-steps require additional cognitive resources, but can also lead to better learning outcomes [3], [9], [58], [75]. Although CLT has been applied in the CS education context [31]–[38], [41]–[43], [76], to the best of our knowledge it has not been previously applied to API learning.

C. External Memory

The idea of an external memory aid has been proposed and studied within the Psychology community. Typically, an external memory aid is created by making a change or changes in an external context (i.e. somewhere other than the person’s brain) to serve as a reminder [21]. Using an external memory aid is one of many forms of cognitive offloading, a process in which a person changes their physical space to lower the cognitive demand of a task [59]. The existing work on external memory aids as reminders does not apply directly to the context of completing tasks using an unfamiliar API. However, one study of when to interrupt programmers observed the use of external memory [17] to manage high cognitive demands.

III. Modeling API Learning

Figure 1 shows an overview of our proposed model of task completion using an unfamiliar API which integrates predictions from Information Foraging Theory (IFT), Cognitive Load Theory (CLT) and the use of external memory aids. The model includes a cognitive context, in which the programmer processes information and makes decisions about which actions to take. To complete a sub-goal, we predict that programmers will move through three stages: Programmers will begin in the Information Collection Stage and interact with the Information Context to find information relevant to their current sub-goal; Next, programmers will move to the Information Organization Stage and interact with the External Memory Context to retrieve and manipulate found information into a usable form; Finally, programmers will move to the
Solution Testing Stage and interact with the Code Context to integrate and test potential solutions to their sub-goal. Below, we describe each of the sub-goal solution stages, the actions that are possible within that stage, and its’ relationship to IFT, CLT, and external memory.

A. Information Collection Stage

Since programmers are initially unfamiliar with the API they are using, they will begin in the Information Collection Stage. In this stage, programmers can search for new information using one of several actions: performing a keyword search, evaluating the results of that search, navigating to a potentially relevant page, and exploring the contents of that page to find potentially useful information.

Search: The programmer performs a new keyword search. At the beginning of the task, the programmer relies on information in long-term memory to select appropriate keywords, leveraging existing knowledge that may not be relevant to the current API. Later on, the programmer may also leverage information found during information collection to select better keywords. In the process of learning how to translate goals into appropriate queries, API specific keywords are a source of extraneous load.

Results: The programmer reviews the results of a recent search, looking for links that may contain relevant information. When reviewing search results, programmers evaluate the likelihood that any given link will contain the target information. We predict that this evaluation will be based primarily on information drawn from long-term memory, short-term memory, and the current search results. Making relevance judgments is a source of extraneous load.

Navigate: The programmer navigates to a new page either from search results or from a currently viewed page. The decision to navigate to a new page is the result of a previous action. The act of navigating does not incur a cognitive cost, though it may incur a time cost.

Page: The programmer searches within the current page for the target information. While viewing a page, the programmer has to scan for potentially relevant sections of information. Determinations of relevance are made by combining information from long-term memory and short-term memory with information on the page itself. Information that the programmer believes to be relevant will be stored in external memory. The degree of information review can vary. The programmer may simply scan for words related to the target information, incurring extraneous load. In other cases, programmers may be unsure of how related the current content is to the target information and read content within the page carefully in order to determine relevance, an activity that may incur extraneous load or germane load, depending on how well the content matches their task. Activities like self-explaining code behavior or reading conceptual material related to the API are more likely to incur germane load.

IFT predicts that programmers will select actions in order to maximize the value of information obtained per cost of interaction [53], [54]. Programmers will continue to collect information and store it using external memory until they believe that they can construct a solution.

B. Information Organization Stage

Once programmers have collected enough information that they believe a sub-goal solution may be possible, they progress to the Information Organization Stage. The goal is to identify relevant information within external memory and use it to compose a potential solution. Programmers often need to integrate information from several sources, and may choose to edit and store partial potential sub-goal solutions in external memory. Along the way, programmers may identify a need to revisit the Information Collection Stage or modify their current sub-goal due to a previously unrecognized information need.

Today’s programmers have little or no explicit support for external memory. Instead, our informal discussions with programmers suggest that they use a variety of techniques, including leaving open web browser tabs or copying relevant information into a code or text editor. These differing techniques may afford different opportunities for integrating and editing the stored information.

Store: The programmer saves relevant information to external memory. This action moves information from Information Collection to Information Organization. Throughout the Information Collection Stage, we expect programmers to store potentially relevant information in external memory for later use. Storing information likely incurs little cognitive cost.

Retrieve: The programmer retrieves information from external memory. Retrieval may include actions like clicking on an open, unfocused tab and scrolling through text in a text editor to find a particular target. In essence, this is an information foraging activity conducted in the external memory context. Relevance determinations are made by combining information on the page with information from short and long term memory, incurring extraneous load. Additionally, programmers may choose to expend additional germane cognitive load to understand content deemed relevant.

Edit: The programmer edits code, either written independently or adapted from one or more examples stored in external memory. In many cases, programmers need to combine information from multiple sources [60], [61] as well as short and long-term memory in order to compose a solution to an identified sub-goal, an activity that can incur extraneous load. Here, external memory serves as an extension of working memory, relieving the programmer of holding all of the solution details in working memory and allowing processing of a single modification at a time.

Delete: The programmer removes information previously stored in external memory. This can occur when a programmer realizes that information previously collected is not relevant to the current sub-goal. Cognitive resources devoted to this information are extraneous. Because this information was organized and elaborated, programmers are more likely to have encoded some of it in long-term memory.

Prior research in CLT suggests that working memory is an extremely limited resource, and one that is easy to overwhelm.
even with carefully designed educational activities [48]. While prior research on external memory focuses on its’ use as a reminder system, researchers have found that the act of organizing it using an external memory can lead to long term learning [21]. Taken together, and within the context of completing a programming task using an unfamiliar API, external memory allows programmers to 1) hold onto relevant information with no cognitive cost, and 2) manage cognitive load associated with integrating information from multiple sources. Further, we predict that information that is actively manipulated via organization or editing (a form of germane load) will be more likely to be encoded to long-term memory.

C. Solution Testing Stage

At the beginning of the Solution their current program.

requirements require the programmer to return to an earlier stage.

of the next sub-goal. However, in some cases, attempting to integrate and test their potential solution. The results of running the program will then inform the programmer’s selection to test. That solution may need to be moved from external sources. Further, we predict that information that is actively manipulated via organization or editing (a form of germane load) will be more likely to be encoded to long-term memory.

C. Solution Testing Stage

At this stage, programmers will have a potential solution to test. That solution may need to be moved from external memory to the code context and modified to fit the current program. In many cases, programmers will be able to easily integrate and test their potential solution. The results of running the program will then inform the programmer’s selection of the next sub-goal. However, in some cases, attempting to integrate the potential solution will result in a question that requires the programmer to return to an earlier stage.

Read Code: The programmer reads existing code in their current program. At the beginning of the Solution Testing Stage, a programmer needs to determine where in their current program code to integrate a potential solution. This early process is information foraging. Later, the programmer may need to re-read code to evaluate its testability.

Edit Code: The programmer edits code to incorporate a potential solution to the current sub-goal. Often, editing will begin with copying and pasting the potential solution from external memory. The programmer may need to modify the code further for use in the current program. In these cases, the programmer will use the code context as an external memory as well as knowledge from long-term and working memory to plan and carry out necessary edits. In other cases, the potential solution may not require modification; pasting a potential solution into the code requires little cognitive investment.

Run Code: Test a solution to determine whether or not it achieves the current sub-goal. Running the code will often be a straightforward action that does not require additional cognitive effort. However, in some cases, the programmer may need to plan interactions with the running program in order to trigger the modified code. While debugging is a complex and interesting activity, we believe it is distinct from API learning and do not focus on it in this model.

Results: Observe the output of the running program to evaluate whether it achieves the sub-goal. In the Solution Testing Stage, the programmer leverages the results of the Information Organization Stage as stored in external memory. The programmer’s IDE can then serve as an additional external memory supporting any further modifications that are necessary to incorporate the solution into the broader context of the program. CLT and external memory research predict that situations in which the programmer has to manipulate a potential solution copied from external memory in order to integrate it are more likely to result in information encoded to long-term memory. This can incur either germane or extraneous load, depending on the nature of the required manipulations. The testing stage does include debugging activities. However, since the focus of this mode is API learning, we only want to model debugging related to misunderstandings of API behavior.

IV. Evaluation

As a preliminary evaluation, we examine our model in the context of two consistent findings in API learning research.

Programmers often struggle to frame questions that address their information need, in part due to difficulties determining appropriate keywords for the target API [14] [78] [64]. Our model suggests that programmers will leverage their existing knowledge or schema in attempting to articulate a question. Therefore, programmers may struggle when the language and structure of the target API does not conform to terminology and concepts they have mastered through the use of other APIs. Designing APIs that match programmers intuitions can be difficult given the diversity of experience they potentially bring to each new API. However, we can begin to design educational scaffolding to facilitate the transfer of concepts and structures from previously used APIs. While some work has begun to explore scaffolding the transfer of programming language knowledge [67], exploration of knowledge transfer for APIs is currently under-explored.

Programmers have difficulty integrating multiple API elements to solve a single problem, suggesting a need to help programmers identify API usage patterns [14] [60] [61]. In order to predict this using our model, it is necessary to know something about the kinds of information that programmers typically find through search. Documentation tends to focus on a single API element at a time [14], while many programming tasks require the integration of multiple API elements. This information split increases cognitive effort programmers must expend to bring the necessary information into a single context. Software tools that more tightly integrate search and development activities and enable easy, simultaneous access to multiple code examples may help to decrease the difficulty associated with the integration of API elements.

V. Conclusion and Future Work

In this paper, we have described a theoretical model of API learning that was derived from three well validated models that describe aspects of the API learning process. Our model is consistent with findings about API learning barriers, providing a preliminary evaluation. Since much of this work is based on data collected after the learning process has concluded, future studies should capture the full process of API learning as a further validation of our model. We are also intrigued by the idea of an external memory that moves easily between information collection and code editing activities. Additional study of the information integration process and the flow from information search to integration could help to inform the design of external memory supports that can go beyond easing the process of re-finding information.
REFERENCES


