View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

Engineering and Applied Science Theses & Dissertations

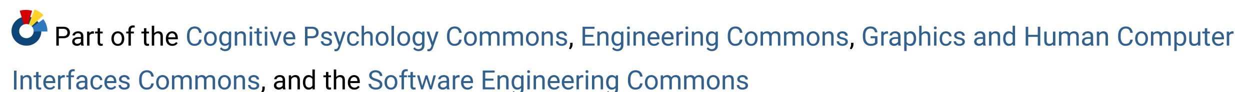
McKelvey School of Engineering

Spring 5-15-2020

Exploring Usage of Web Resources Through a Model of API Learning

Finn Voichick

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds


Part of the Cognitive Psychology Commons, Engineering Commons, Graphics and Human Computer Interfaces Commons, and the Software Engineering Commons

Recommended Citation

Voichick, Finn, "Exploring Usage of Web Resources Through a Model of API Learning" (2020). *Engineering and Applied Science Theses & Dissertations*. 513.
https://openscholarship.wustl.edu/eng_etds/513

This Thesis is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Caitlin Kelleher, Chair
Alvitta Ottley
Dennis Cosgrove

Exploring Usage of Web Resources Through a Model of API Learning

by

Finn Voichick

A thesis presented to the McKelvey School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

May 2020
Saint Louis, Missouri

Copyright © 2020 Finn Voichick

This work is licensed under the
Creative Commons Attribution-ShareAlike 4.0 International License.
To view a copy of this license, visit
https://creativecommons.org/licenses/by-sa/4.0/.

Contents

List of Tables . . . . **iv**

List of Figures . . . . **v**

Acknowledgments . . . . **vi**

Abstract . . . . **vii**

1 Introduction **1**
1.1 Background 1
1.1.1 Api Usability 2
1.1.2 Information Foraging 2
1.1.3 Cognitive Load 2
1.1.4 External Memory 3
1.2 The Coil Model 3
1.2.1 Information Collection 4
1.2.2 Information Organization 5
1.2.3 Solution Testing 6
1.3 Research Contributions to the Field 6

2 Logging System **8**
2.1 Motivation 8
2.1.1 Screen and Audio Recordings 8
2.1.2 Choice of Editor and Browser 9
2.2 Architecture 10
2.2.1 Atom Package 10
2.2.2 Chrome Extension 11
2.2.3 Chromium Modification 14
2.2.4 Other Scripts 15
2.2.5 Shortcomings 15

3 Model Study **17**
3.1 Experimental Setup 17
3.1.1 Participants 17
3.1.2 Nasa Task Load Index 18

3.1.3 Development Environment . . . 18
3.1.4 Task . . . 19
3.1.5 Pilot Study . . . 20
3.2 Results and Analysis . . . 21
3.2.1 Actions Performed . . . 21
3.2.2 Types of Pages Viewed . . . 22
3.2.3 Sequences of Actions . . . 23
3.2.4 Tab Creation and Deletion . . . 24
3.2.5 Tab Retrieval . . . 26
3.2.6 Changes Made to Pasted Blocks . . . 26
3.2.7 Unsuccessful Pastes . . . 28
3.2.8 Cognitive Load . . . 29

4 Conclusions . . . 30
4.1 Discussion . . . 30
4.1.1 Relevance of External Memory . . . 30
4.1.2 Sources of Cognitive Load . . . 31
4.2 Future Work . . . 32
4.2.1 Expanded Study . . . 32
4.2.2 Measures of Cognitive Load . . . 32
4.2.3 External Memory Management Tools . . . 32

References . . . 34

List of Tables

2.1 Atom Package Logged Events 11
2.2 Chrome Extension Background Script Logged Events 12
2.3 Chrome Extension Content Script Logged Events 13
2.4 Chromium Modification Logged Events 14

List of Figures

1.1 The Coil Model of Api Learning 4

3.1 User Study Protocol 18
3.2 Page Types Viewed Over Time 23
3.3 Common Transitions Between Actions 24
3.4 Tabs Open Over Time 25
3.5 Transitions Between Edit Actions 27
3.6 Edit Actions Preceding Browser Actions 28
3.7 Ratings of Mental Effort and Frustration via the Nasa-Tlx 29

Acknowledgments

I want to thank Caitlin Kelleher for advising me in this research. Her focus, patience, and expertise have been extremely valuable during my first couple years as a researcher, and I'm sure they've had a hugely positive impact on my future research career. Her thoughtful answers to all of my questions have helped shape the way I think about research, and for that I am very grateful.

I would also like to thank Michelle Ichinco and Gao Gao, the other great members of our research team. They've been a pleasure to collaborate with.

I'd like to thank Dennis Cosgrove for being a great mentor throughout my time at WashU. He encouraged me early on in my undergraduate career to get involved in research, and the enthusiasm that he brings to everything he does has made it really fun to be his teaching assistant.

Finally, I would like to thank my parents and my fiancée Abby for their unwavering support of me and my interests. Their encouragement has been very valuable.

Finn Voichick

Washington University in Saint Louis
May 2020

ABSTRACT OF THE THESIS

Exploring Usage of Web Resources Through a Model of API Learning

by

Finn Voichick

Master of Science in Computer Science

Washington University in St. Louis, May 2020

Research Advisor: Professor Caitlin Kelleher

Application programming interfaces (APIs) are essential to modern software development, and new APIs are frequently being produced. Consequently, software developers must regularly learn new APIs, which they typically do on the job from online resources rather than in a formal educational context. The Kelleher–Ichinco COIL model, an acronym for "Collection and Organization of Information for Learning," was recently developed to model the entire API learning process, drawing from information foraging theory, cognitive load theory, and external memory research. We ran an exploratory empirical user study in which participants performed a programming task using the React API with the goal of validating and refining this model. Our results support the predictions made by the COIL model, especially the role of external memory in the API learning process. Participants extensively used browser tabs to store web resources in external memory, but their behavior suggests some inefficiencies that incur extraneous cognitive load.

Chapter 1

Introduction

Much programming by modern software developers is achieved through the use of application programming interfaces (APIs). These software elements abstract away the details of a complicated underlying system, allowing developers to reuse code and interact with large software systems relatively easily. New APIs are constantly being announced [9], and software developers today may program almost entirely using APIs [23].

For these reasons, the ability to learn new APIs has become a valuable skill for software developers. However, this learning is often done on the job rather than in a formal educational setting [12]. Researchers have tried to understand and improve this learning process in various ways. The work described in this thesis focuses on the Coil model [20] (described in Section 1.2), which is grounded in several theoretical areas and attempts to holistically model the API learning process.

Our research team ran an exploratory empirical lab study in an attempt to validate and refine the Coil model. Our findings help to validate the model, especially the critical role played by external memory resources. Results also point toward some inefficiencies in the API learning process that future work should aim to mitigate.

1.1 Background

This work complements existing work in the fields of API usability, information foraging theory, cognitive load theory, and external memory.

1.1.1 API Usability

The field of API usability draws from various human-computer interaction methods to make APIs easier for developers to use. Work in this area has taken various approaches. Some have focused on the experiences of those using existing APIs, using methods such as surveys [28] and interviews [17]. These studies have uncovered some common issues faced by developers, such as integrating code involving separate API components [14]. Other studies have focused on the design of APIs, using empirical studies to develop API design guidelines [26] or elicitation studies to guide API design [18]. Still other studies have focused on ways to improve API documentation [22]. These studies have provided valuable insights, but have not attempted to model the API learning process as comprehensively as the COIL model.

1.1.2 Information Foraging

Information foraging theory explains how people use the web to search for information. It models the process on the way that animals forage for food [27]. Web users adaptively judge the relevance of the information they see using "information scent" [13] to seek out valuable "information patches." This theory has been applied to several aspects of the software engineering process, such as debugging [16] and code navigation [24, 21].

We expect information foraging theory to be readily applicable to the API learning process. Much of API learning depends on the developer's ability to find the relevant information online, and the finding of this information is well-modeled by information foraging theory.

1.1.3 Cognitive Load

Cognitive load theory describes the role of working memory in learning processes. According to cognitive load theory, working memory resources are often a bottleneck. There are three types of cognitive load involved in a learning task: intrinsic load, extraneous load, and germane load [29]. Intrinsic load is related to the difficulty of the learning task and the learner's experience, and is not considered changeable. Extraneous load arises when instructional materials are inefficient, for example when information is redundantly presented or a learner

must mentally integrate multiple sources of information [29]. Germane load is invested by learners when they put in extra effort to help learn the material at hand, for example by describing the principles involved in a procedure that they are trying to learn [10]. Because cognitive load resources are limited, a general goal for designers of instructional material is to reduce extraneous load, allowing for investment of germane load.

1.1.4 External Memory

External memory is the use of aids external to oneself to facilitate later retention of information [19]. For example, sticky notes and calendar reminders are external memory aids used commonly in day-to-day life. Little work has been done around the role of external memory in programming tasks. Some have pointed to the use of the web as an external memory resource, where programmers don't remember specific syntax but they remember a webpage that had the needed information, retrieving the information from the web when needed [11]. The COIL model below identifies a much more common use of external memory: browser tabs which store webpage information.

1.2 The COIL Model

The COIL model [20], an acronym for "Collection and Organization of Information for Learning," draws from information foraging theory, cognitive load theory, and external memory research to model the process of learning an API. It groups actions taken by API learners into three stages: information collection, information organization, and solution testing. It predicts that learners will take certain actions in each stage and that the entire process will be mediated by cognitive load. Figure 1.1 shows a graphical representation of the COIL model.

The model predicts that when a developer is working on a subgoal, they will generally progress from the information collection stage to the information organization stage and then to the solution testing stage. However, this order is not rigid, and developers may move

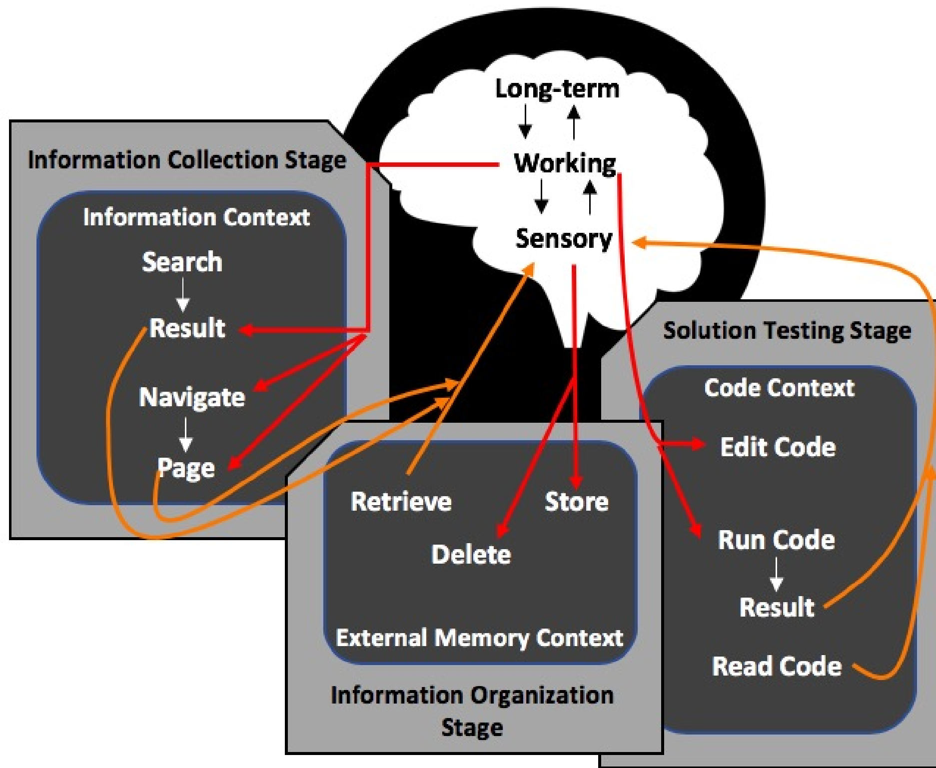


Figure 1.1: The COIL Model of API Learning

back and forth through stages depending on their progress. I will now discuss each stage and its associated actions.

1.2.1 Information Collection

The information collection stage is the first stage in the model, and it describes the way that developers gather information about the new API. Developers generally start in this stage when working on a subtask, and may return to this stage from others if they realize a need for additional information. This stage has four general actions:

- Perform a keyword **search**. This is the starting point for the gathering of information, with keywords coming from long-term memory or from recently-found information.

- Review search **result**s. Developers evaluate the relevance of search results from a mix of long-term memory and recently-gathered information.
- **Navigate** to a new page. This can happen either from the search results page or from a previously-found webpage.
- Search for information within the current **page**. The developer may scan the page for relevant words, or may read in depth.

The behavior in the information collection stage is largely explained by information foraging theory. Throughout, developers must make relevance judgements based on information scent, leading them to useful webpages. There is plenty of opportunity for extraneous and germane cognitive load throughout this stage as well. Developers may incur extraneous load if the API uses special keywords that aren't easily searchable, or if they read a webpage that turns out to be redundant with information they already know. On the other hand, a developer who reads conceptual information related to an API may be investing germane load.

1.2.2 Information Organization

The information organization stage describes how developers store and retrieve information collected in the information collection stage. Many aspects of this stage are explained by external memory research, as it describes how developers manipulate their external memory resources. The information organization stage has three general actions:

- **Store** information to external memory. Developers will transition from the information collection stage to the information organization stage by saving the information in an external memory context.
- **Retrieve** information from external memory. This action is essentially foraging for information that is stored in external memory.
- **Delete** information stored in external memory. This is done when stored information is deemed irrelevant.

A crucial role of the information organization stage is to alleviate the cognitive cost of keeping track of information found in the information collection stage. Additionally, the act of organizing and judging the relevance of stored information can be a form of germane load.

1.2.3 Solution Testing

The solution testing stage describes how developers integrate information into their program. Developers typically arrive at this stage after information organization. This stage has four general actions:

- **Read code** in the current program. This step may involve program comprehension to determine the relevant location in the code.
- **Edit code** to incorporate a potential solution. This step may involve code directly copied from external memory and/or smaller modifications.
- **Run code** to test it.
- Observe the **result**s of the running program. The results will determine whether changes must be made (potentially involving a return to a previous stage) or whether the developer can progress to a different subtask.

The cognitive load required in the solution testing stage depends largely on the "edit code" step. This step may be a simple matter of copy-and-paste, or it may require more extensive modifications with a high cognitive cost.

1.3 Research Contributions to the Field

The COIL model as described in Section 1.2 was developed by Kelleher and Ichinco [20]. Our goal was to study real programmers learning a new API in order to validate and refine this model, which required us to run a user study.

The user study required the development of a logging system that could capture all of the relevant actions taken by our participants. I developed this logging system (described in Chapter 2) that logs a range of actions relevant to our model across both the web browser and code editor.

The user study provided some interesting results, described in Chapter 3. For one, participants did perform the actions predicted by the Coil model, providing some validation for the model. Additionally, participants who interacted more with code that they pasted into their program were less likely to later delete the pasted code, suggesting that investing extra effort into program comprehension can lead to a decrease in later extraneous cognitive load. Participants extensively used browser tabs as a key context for external memory, but some behaviors pointed to inefficiencies in external memory retrieval, indicating a need for better external memory management tools.

Chapter 2

Logging System

2.1 Motivation

Our study, described in depth in Chapter 3, required participants to complete a programming task. Participants were allowed to browse the web for help in completing this task, and we aimed to find and characterize patterns in behavior across code editing and web browsing. This required us to collect a wide range of data related to user actions in their code editor and in their web browser.

2.1.1 Screen and Audio Recordings

For all of our participants, we recorded the screen as they were performing the task. In addition, a microphone was used to record audio for the think-aloud protocol. These recordings were useful; however, they were insufficient for collecting the volume of data that we needed.

Manually transcribing the required information from screen and audio recordings alone would not have been feasible. Participants frequently performed a sequence of actions in rapid succession (for example, switching between tabs, copying, pasting, and typing), and it was necessary to precisely record the timing of all of these events. For this reason, we needed a system to automatically record participant actions as they were being performed.

2.1.2 Choice of Editor and Browser

The most straightforward way to build the logging system was to take advantage of the developer APIs provided by popular editors and browsers. These APIs vary among different browsers and editors, so it was necessary to select a specific editor and browser for participants to use. We settled on the Atom text editor and the Chromium browser.

We chose to require participants to use the Atom text editor [1], developed primarily by GitHub. While it is difficult to find an unbiased listing of which editors are commonly used with React programs, simple web searches like "best react editors" frequently result in lists of editors that include Atom, and it is a fairly popular editor overall [7], so it is reasonable that a React newcomer would use Atom. In addition, Atom's package API provides all of the needed functionality, and there are a wide range of existing free and open-source Atom packages for reference. As an added benefit, JavaScript is the language used to develop Atom packages, which was required for the browser extension, allowing code re-use across different components of the logging system.

The Chromium browser [3], developed primarily by Google, was chosen primarily because it provided a good mix of familiarity and adaptability. Google Chrome is by some measures the most popular web browser [4], and so we expected our participants to be generally familiar with the features provided by Google Chrome. This was beneficial because we wished to make the setting for study to be as realistic as reasonably possible, and we didn't wish to confound our results with users struggling to understand the features of an unfamiliar web browser. In addition, I found Chrome's extension API [2] fairly well-documented and easy to use, which made it feasible to develop part of the logging system as a Chrome extension.

Unfortunately, there were some actions that we wanted our logging system to capture, but that were not feasible with Chrome's extension API. For example, while Chrome's extension API provides a "`chrome.tabs.onCreated`" event that is triggered whenever a new tab is created, there is no such event for when a user clicks the "back" button or opens the "find in page" search bar. To capture these events, we had to make modifications to the browser itself. This was not feasible within the proprietary and closed-source Google Chrome browser, and so we decided to use Chromium, the free and open-source web browser upon which Google Chrome is based. Chromium is nearly identical to Google Chrome for the typical user.

2.2 Architecture

The logging system I developed has three primary components: a GitHub Atom package, a Google Chrome extension, and a Chromium modification. These components all intercept participant actions and save related information for later analysis.

2.2.1 Atom Package

The Atom package was created using Atom's package API. The API provides events to which packages can subscribe with a callback function. When certain events are triggered, our callback function is called, which uses Atom's API to save a JSON file to a directory hidden to the participant. To avoid using too much memory, each action is saved as a separate file as it occurs. All files saved included an action identifier and a timestamp for the event, and most logged events also included additional action-specific information. Table 2.1 provides a list of all seventeen events logged by our Atom package and the action-specific information logged with each.

During a typical editing session, the "scroll" and "highlight" events often fire many times in rapid succession. For example, if a user scrolls from the top of the file to the bottom, the scrolling is internally treated as many "steps," and the scroll event is fired at each step. For performance reasons, the "scroll" and "highlight" events were debounced such that two scroll events or two highlight events within 0.5 seconds of each other would be treated as a single event, and only the final scrolled position would be logged.

In Atom, it is possible to bring up a "Find in Buffer" pop-up using either a menu or the Ctrl+F keyboard shortcut. This functionality allows for simple searching and refactoring, and it is possible through a "find-and-replace" package [5] that is installed by default with Atom. However, there are no events for our package to hook into to cause our callback function to be triggered when a participant uses this functionality. The find-and-replace package is free and open-source, so I modified it to save files with relevant information when performing the relevant actions, replacing the find-and-replace package on the test computer with my modified version. Thus, the "Atom package" component of the logging system is

Table 2.1: Atom Package Logged Events

Logged Action	**Additional Logged Information**
switch focus to a different file	new file contents (full and visible)
change a file in any way	text added and removed, location in file
stop editing for at least 300ms	full file contents
move or rename file	new file path
save file through Ctrl+S or menu	full file contents
vertically scroll	visible lines in file
read from clipboard (paste into editor)	text read (pasted)
write to clipboard (copy from editor)	text written (copied)
highlight or unhighlight text	selected text
open "Find in Buffer" pop-up	n/a
type in "Find in Buffer" pop-up	typed text
"Find Next" in "Find in Buffer" pop-up	searched text
"Find Previous" in "Find in Buffer" pop-up	searched text
"Find All" in "Find in Buffer" pop-up	searched text
"Replace Next" in "Find in Buffer" pop-up	searched text, replacement text
"Replace Previous" in "Find in Buffer" pop-up	searched text, replacement text
"Replace All" in "Find in Buffer" pop-up	searched text, replacement text

actually two Atom packages: the modified find-and-replace package and the package that logs all of the other actions.

2.2.2 Chrome Extension

Most browser actions were logged with a Chrome extension, which was added to Chromium on the test computer. The Chrome extension was built using Chrome's extension API, and it captures participant actions in two different ways: a "background scipt" and a "content script," both written in JavaScript. Together, these scripts logged twenty-four types of actions, listed in Tables 2.2 and 2.3.

Unlike Atom's package API, neither Chrome's extension API nor the JavaScript language has functionality for direct file manipulation. Chrome's extension API does include a "`chrome.storage`" API for saving information locally on a user's machine, but I found that several bugs within Chrome prevented this system from storing the amount of log data

needed without crashing. As a workaround, I created another program that simply listened for HTTP requests on a free port on the local machine and saved any received data as a JSON file. I configured Chromium to automatically run this program on startup, and then the Chrome extension was made to send any data it wanted to save through an HTTP request to this other program, allowing log files to be saved as desired.

The background script is designed to log the sixteen browser-level actions listed in Table 2.2 (as opposed to page-level actions). These are all events provided by Chrome's extension API, and involved manipulation of windows, tabs, and bookmarks. All of these logs include an action identifier and a timestamp, as well as other action-specific information listed in Table 2.2.

Table 2.2: Chrome Extension Background Script Logged Events

Logged Action	**Additional Logged Information**
open new window	window ID, complete tab listing
close window	window ID, complete tab listing
window focus change	newly-focused window ID, complete tab listing
open new tab	tab ID, complete tab listing
update tab properties (e.g. URL)	tab ID, complete tab listing
move tab within window	tab ID, complete tab listing
activate (focus) tab	tab ID, complete tab listing
detatch tab from window	tab ID, complete tab listing
attach tab to window	tab ID, complete tab listing
close tab	tab ID, complete tab listing
zoom in/out in tab	tab ID, complete tab listing
create bookmark	title, url, bookmark ID, folder
remove bookmark	title, url, bookmark ID, folder
change bookmark	title, url, bookmark ID
move bookmark	title, url, bookmark ID, folder
reorder bookmarks	folder ID, new order

The content script is designed to log the eight page-level actions listed in Table 2.3. This script was inserted into every webpage visited by participants, and so it had access to all HTML DOM events that a webpage's own scripts would have access to. The "highlight" and "scroll" events are debounced in the same way as in the Atom package.

Table 2.3: Chrome Extension Content Script Logged Events

Logged Action	**Additional Logged Information**
load page contents from web	all HTML on page, visible HTML on page
highlight text	selected text
cut text from webpage	cut text
copy text from webpage	copied text
paste text into webpage	pasted text
scroll within webpage	visible HTML on page
click middle mouse button	n/a
React error message	HTML of message

For two of these actions, "load page" and "scroll," it was necessary to log the content that was currently visible on the page. This was done by copying the HTML in the page and then recursively deleting HTML elements with bounding boxes that did not overlap with the viewport. This method usually worked, but sometimes removed visible content on pages with more complicated CSS styling.

Another issue was encountered when copying from webpages. For security reasons, Chromium does not allow scripts on webpages (including content scripts) to read the clipboard except when the user is pasting into the webpage, or after a warning message is shown to the user about the page accessing their clipboard. Such a message would have been too disruptive if it occurred every time a participant tried to copy text from a webpage. As a workaround, when the user copies text from a page, the content script records which text is currently highlighted in the page.

Participants tested their React application within a browser tab, so interaction related to testing was captured by the Chrome extension (and Chromium modification). The React testing page was set up to automatically refresh its content when a user saved edits to files. (Section 3.1.3 describes the development evironment that participants used.) To check for React error messages, the Chrome extension listened for mutations made to the content in the testing tab that looked like error messages.

2.2.3 Chromium Modification

Chromium source code was modified to log certain events. This was done by searching through the Chromium codebase for methods that are called when specific actions are performed, and then inserting code that logged relevant information: an action identifier, timestamp, and potentially action-specific information, listed in Table 2.4. As seen in the table, most of the actions did not log additional information.

Table 2.4: Chromium Modification Logged Events

Logged Action	**Additional Logged Information**
show main menu	n/a
show submenu	name of submenu
click "Back" button	n/a
click "Forward" button	n/a
open back/forward dropdown	n/a
restore tab from history	n/a
open "Find..." bar	n/a
find in page	text searched for, direction searched
read (paste) text from clipboard	read (pasted) text
enter full-screen mode	n/a
save page to file	n/a
zoom in on page	n/a
zoom out on page	n/a
reset zoom on page	n/a
create website shortcut as file	n/a
open "Developer tools"	n/a
toggle "Developer tools"	n/a
open developer console	n/a
message appears in developer console	text of message
toggle developer "device toolbar"	n/a
open developer "Inspect" menu	n/a
open Chromium task manager	n/a
show bookmarks bar	n/a

Most of the Chromium source code is written in C++, so the inserted code was generally fairly straightforward. However, for certain actions, like "message appears in console," the relevant code was written in JavaScript. For these actions, I re-used code from the Chrome extension that sends the data over HTTP to the file-writing program.

2.2.4 Other Scripts

The components of the logging system are all designed to save files to a particular folder on the hard drive. This made the components easier to write; for example, the Chromium modification was just a few lines of code pasted into several parts of the Chromium codebase. However, it makes it more difficult to organize the files by participant. To solve this problem, I created a script that creates a symbolic link from the log-saving folder to a designated participant-specific save folder.

As a participant uses Atom and Chromium, a log file is saved individually for each action performed. The Atom package and Chrome extension save JSON files, while the Chromium modification saves TXT files. For ease of file transfer and later analysis, I also created a script that consolidates all of these separate files into a single large JSON file. In addition, I created a version that groups the consolodated logs into consecutive browser actions and consecutive editor actions to allow us to visually see web sessions and editor sessions when looking through the log files.

2.2.5 Shortcomings

There were some actions that the logging system should have captured but regrettably didn't. Most notably, there is no logged event for when a user performs an undo or redo action. More specifically, the logging system logs every change made to a file, but doesn't specifically note if the cause of that change was an undo or redo action.

In our later analysis, I had to get around this problem by implementing an "undo" and "redo" stack and using them to infer when a change was caused by an undo or redo action. This worked, but required more effort than it would have taken to implement undo/redo tracking into the logging system.

Additionally, it may have been useful to have a keylogger running while participants were working on the programming task. There were some events within Atom that could be triggered either by a menu action or by a keyboard shortcut, and it might have been useful to know which was used. A keylogger would have captured a lot of user input not captured by

the rest of the logging system, and would also have made it easier to distinguish undo/redo actions.

Chapter 3

Model Study

After building the logging system discussed in Chapter 2, we conducted a user study in which we asked React newcomers to perform a programming task using the React API.

3.1 Experimental Setup

3.1.1 Participants

Not counting pilot participants (discussed in Section 3.1.5), fourteen participants were recruited through a computer science department mailing list, and all had some programming experience. There was a technical issue with the logging system that prevented us from collecting data from one participant, so we report on the other thirteen participants. Their ages ranged from 19 to 34 years, with a mean of 21.9 and a standard deviation of 4.0 years.

All participants had experience with at least one programming language. Five participants had experience with neither React nor JavaScript, three participants had experience with both React and JavaScript, and the remaining five participants had experience with JavaScript but not React.

3.1.2 NASA Task Load Index

To measure cognitive load, we used the NASA Task Load Index [6] (NASA-TLX), a standard measure. This questionnaire asks participants to rate six aspects of task workload: mental demand, physical demand, temporal demand, performance, effort, and frustration. Participants rate each aspect on an individual scale, and also rank which aspects had the most impact on the overall workload.

Before working on the programming task, participants were familiarized with the NASA-TLX with a five-minute warm-up task that asked them to identify several locations from vague references that were not easily searchable. This practice task was designed to give users practice with the rating scales. During the one-hour programming task (described in Section 3.1.4), participants were interrupted three times at twenty-minute intervals to complete the NASA-TLX, ranking their workload over the last twenty minutes. See Figure 3.1 for a graphical representation of the task timeline.

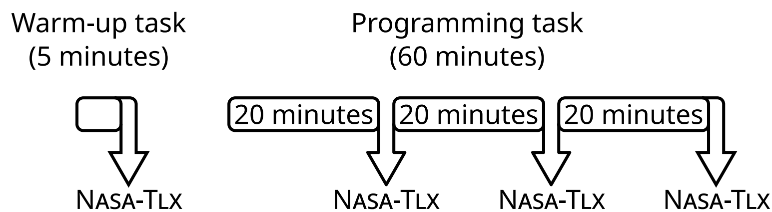


Figure 3.1: User Study Protocol

3.1.3 Development Environment

Participants were provided with a working React environment created using Facebook's "Create React App" package [15]. This package is suggested in the official "Intro to React" tutorial [8]. It includes a minimal single-page React application, and it is set up so that whenever a user saves a change to one of their code files, the application is automatically reloaded in the web browser.

Participants were provided with a computer that had two open windows (one window for the Atom text editor and one window for the Chromium web browser) and all components of the logging system were installed and running. In the Atom window, the sample React

project was open. A text file was added to the sample project that contained a description of the programming task they were asked to complete, described in Section 3.1.4. The Chromium window had a single tab open: the live sample application. This setup was meant to minimize any issues with the development environment; participants generally found it fairly straightforward to modify files within the Atom window, save their changes, and then immediately see changes in the application tab within the Chromium window.

3.1.4 Task

Participants were given one hour to complete a task using the React API. They were asked to follow a think-aloud protocol, verbalizing their thought processes as they worked on the task. The task description file provided to participants was as follows:

```
1
2
3 Modify this React JS application so that it has two basic controls: a text
     box and a button labeled "enter". The button should be disabled
    initially, but should be enabled when the user types "friend" into the
    text box.
4
5 http://localhost:3000/ shows the running result. Save your file to see
    your changes.
```

This task was chosen for several reasons:

- It is understandable. Developers tend to be familiar with text boxes and buttons, and the behavior of the text box and button in response to user input is straightforward.
- It is simple. The whole task is two sentences and there are few moving parts, so it is relatively easy for a participant to work on the task without forgetting their goal.
- It avoids hints. The desciption is given in plain English and doesn't give implementation details away, like the fact that a "text box" is called an "input" in HTML.
- It is divisible into subtasks. Participants were able to track their progress as they completed different parts of the task, like showing a button on-screen with the correct text.

- It is nontrivial. This task takes some time to complete, and there is no code readily available online that solves this particular problem.
- It requires integration of several resources. Participants need to consult several different help resources for the different aspects of this task and combine them in a logical way, something that is notoriously difficult for API newcomers.
- It is challenging but possible to complete in an hour. Most participants spent the entire hour working on the task, giving us a sufficient amount of data, but participants were generally able to complete at least part of the task.

3.1.5 Pilot Study

We recruited several pilot participants to complete the task. I recruited three pilot participants in St. Louis, and our colleagues in Massachusetts recruited five. Based on our observations of the pilot participants, we made several minor changes to the user study.

Our pilot participants helped us to refine our React programming task. We originally considered giving our participants a second task in addition to the one described in Section 3.1.4, but that task proved challenging (and time-consuming) enough on its own, so we felt it was sufficient. In addition, the language of the task description originally read to "Build an application" but this may have caused some confusion about what participants were required to do. One participant seemed not to realize that they were given template code and seemed to be trying to build a new application from scratch, so the language was changed to "Modify this ReactJS application."

The pilot participants also helped us make additions to the logging system (described in Chapter 2). There were several actions taken by participants that we did not originally anticipate, but that we felt were relevant to our study. For example, one pilot participant used the mouse wheel "middle click" button within Chromium to open a page in a new tab, leading me to add that action to the logging system.

Finally, after the study was complete, we used data from the pilot participants to guide our analysis. We didn't know exactly what results would be interesting, but we didn't want to

bias our search by seeing the participant data. For this reason, we used the pilot participant data to explore trends and look for interesting results, but the results reported in Section 3.2 are from the normal participants, with pilot participants omitted.

3.2 Results and Analysis

3.2.1 Actions Performed

The Coil model, described in Section 1.2, predicts several actions that users will perform when learning a new api. All of our participants performed all of these actions, though they did so in different ways.

- **Search**. All participants performed a search query using a search engine. One participant used only the Stack Overflow website for these searches, but all other participants used the Google search engine, the default search engine in Chromium.
- Review search **result**s. All participants interacted with a search results webpage in some way, via scrolling and/or clicking on a search result.
- **Navigate** to a new page. All participants viewed a webpage that was neither a search result page nor their localhost testing page. The types of pages viewed are described in Section 3.2.2.
- Search a **page** for target information. All participants interacted with a webpage that was neither a search result page nor their localhost testing page.
- **Store**. All participants copied code snippets from webpages into their editor, either automatically with copy-and-paste functionality, or by typing, visually copying from an open webpage. Ten out of thirteen participants used both copy-and-paste and typing to copy code, two participants used only typing, and one participant used only copy-and-paste. Additionally, all participants kept multiple tabs open, storing previously-visited webpages. See Section 3.2.4 for more information about tab use.

- **Retrieve**. All participants revisited webpages they had previously stored in an unfocused tab. See Section 3.2.4 for more information.
- **Delete**. All participants deleted stored information in some way, either by closing an open window or tab, or by deleting a copied code snippet.
- **Read Code**. All participants shifted focus to Atom and read their code.
- **Edit Code**. All participants changed their code in some way, either by manually typing or by copying and pasting.
- **Run Code**. All participants saved changes to their code. The React environment given to participants (described in Section 3.1.3) was set up so that whenever a file was saved, the localhost testing tab would refresh, in effect "running" their code. Of the times that a file was saved, it was unclear how many times a participant intended to "run" the code rather than simply save it.
- Check **result**s. All participants activated the localhost testing tab.

3.2.2 Types of Pages Viewed

Participants used a range of online resources when working on the task. Ignoring the localhost testing tab, we categorized all webpages visited into six categories:

- Search results: results of queries on search engines like Google and Stack Overflow.
- Official documentation: information about React provided by ReactJS.org or Facebook.
- Unofficial reference: tutorials, blogs, and reference materials written by other sources, such as W3Schools.
- Q&A forum: sites like Stack Overflow where questions are posted and answered.
- Emulator: interactive online editors with testable example code.
- Video: video tutorials on websites like YouTube.

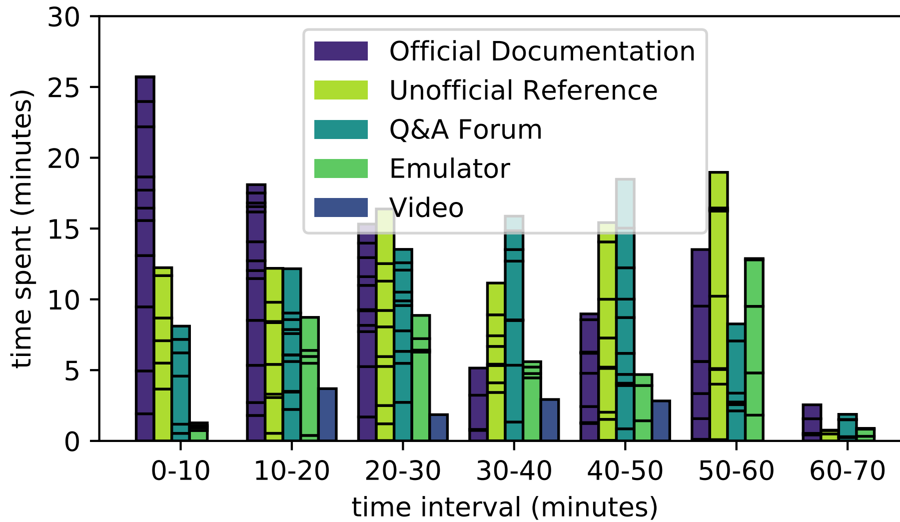


Figure 3.2: Page Types Viewed Over Time

Figure 3.2 shows the types of non-search pages that participants viewed over the course of the task. Each vertical bar shows the amount of time spent across all participants viewing a page of a particular type during a given ten-minute interval, and each bar is segmented by individual participants. "Time spent" is measured as the amount of time that a page of a given type is in an activated tab within a focused Chromium window. As can be seen, participants tended to rely more on official documentation early in the task, but on unofficial references later in the task.

3.2.3 Sequences of Actions

Figure 3.3 shows the transitions that participants made between various actions. Actions are represented as boxes, grouped by the stage in the model. Actions shown are "search" (loading a search query page), "new page" (loading a non-search webpage that had not previously been seen), "new tab" (opening a new tab), "close" (closing a tab), "organize" (rearrange tabs), "return" (view a non-search webpage that had previously been seen), "back" (use the browser's back button or history functionality), "copy" (copy text from a webpage), "edit" (change a file in Atom), and "test" (save and run their code). Arrows show transitions that participants performed at least once on average, exiting the bottom of the source action and

entering the top of the destination action, with thicker arrows representing more actions performed more frequently. As seen in the figure, some of the most common transitions were from a search to a new page (within the information collection stage), between editing and testing in both directions (within the solution testing stage), and from editing or testing to returning to a previously-viewed page (transitioning from solution testing to information organization).

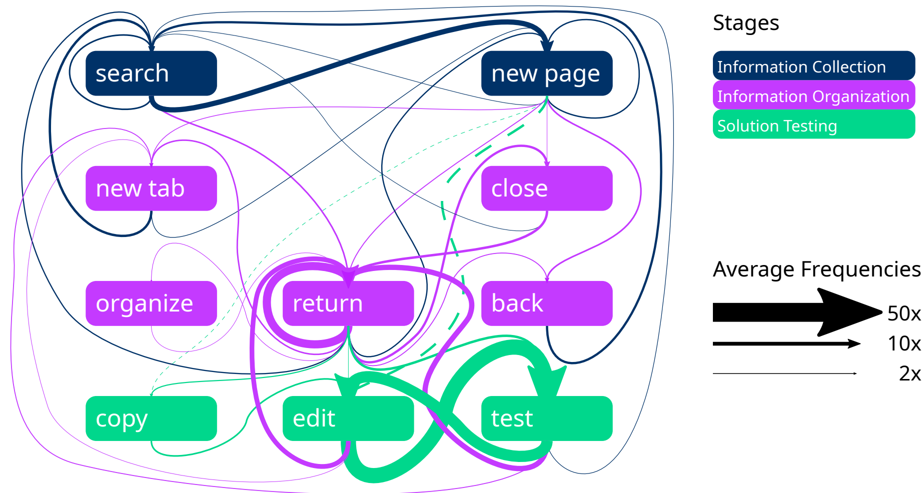


Figure 3.3: Common Transitions Between Actions

When participants transitioned out of the information collection stage, it was often immediately after viewing a new page. When this happened, participants typically transitioned into the information organization stage, but occasionally skipped this stage and went straight into the solution testing stage (represented by dashed arrows in Figure 3.3). However, these transitions were relatively rare.

3.2.4 Tab Creation and Deletion

Participants made heavy use of browser tabs to store pages for later reference. The number of new tabs created by participants ranged from 8 to 39, averaging 21.1 with a standard deviation of 9.1 tabs. However, participants generally closed far fewer tabs during their sessions. Only nine out of thirteen participants closed a tab that they had opened, and they only closed an average of 6.7 with a standard deviation of 7.7 tabs.

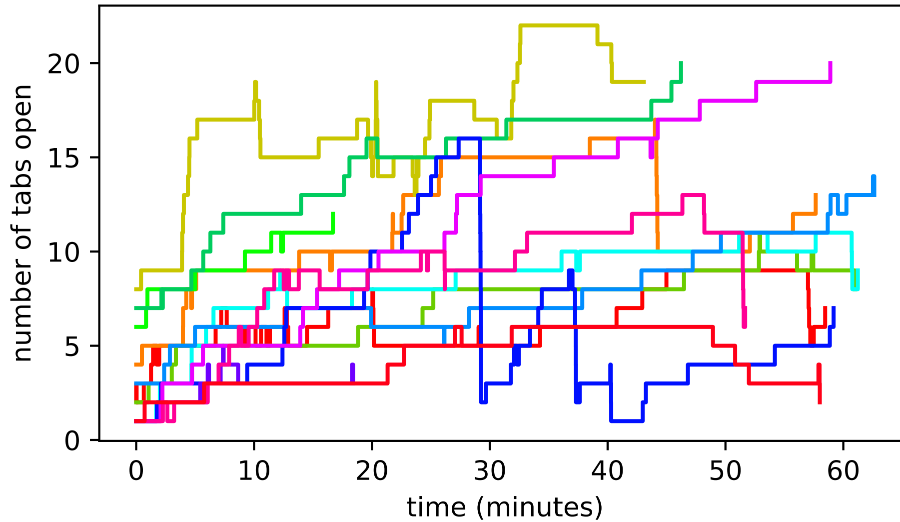


Figure 3.4: Tabs Open Over Time

Figure 3.4 shows each participant's number of tabs open over the duration of the task. Each colored line in the figure corresponds to a different participant. As can be seen, there is a general upward progression as participants tend to open more tabs than they closed. Those participants that did close tabs often closed multiple tabs in rapid successsion, as can be seen for example by the vertical blue line just before 30 minutes.

This behavior may be explained by a relatively high cognitive load cost in evaluating the relevance of a webpage. Determining the usefulness of a web resource may be challenging and a waste of valuable cognitive resources. Participants may continue to open tabs until the cost of organizing a large number of tabs grows unmanageable. At that point, it is worth investing the mental effort required to determine which resources are irrelevant, and the participant is able to judge several tabs at a time.

The reluctance to close tabs could also be explained by a high cost of retrieving a closed tab when it later is found to be relevant. In fact, this turned out to be a fairly common occurence; 22.6% of the times that a participant clicked on a search result, it was a page that the participant had already viewed.

3.2.5 Tab Retrieval

The fact that participants created significant numbers of tabs is not in itself evidence that tabs were being used as external memory. That evidence comes from the fact that participants frequently returned to tabs that had previously been deactivated. The average participant returned to 68% of all visited webpages (with a standard deviation of 14 percentage points), and the average viewed page was returned to an average of 4.3 times (with a standard deviation of 5.7 times). An analysis of the time spent browsing the web across all participants revealed that only 48% of the time was spent searching and visiting new pages. Together, these results indicate that browser tabs served as a vital external memory context.

Figure 3.3 shows that it was relatively common for a participant to transition from viewing a previously-viewed page to viewing a different previously-viewed page. Interestingly, when the average participant made this transition, there was a 71.1% chance that they did so after viewing the page for less than five seconds (with a standard deviation of 20.4 percentage points). This indicates that "flipping through" multiple tabs to find the desired stored information was a common occurance.

3.2.6 Changes Made to Pasted Blocks

Most participants (ten out of thirteen) used copy-and-paste functionality to transfer at least one full line of code from a snippet online into their editor. We were interested in what participants did with these pasted blocks of of code after pasting them.

We categorized pasted code blocks (where a "block" is at least one full line of code) into two categories: "whole" (where an entire code example is copied) and "partial" (where only some of the lines are copied). We further separated these pastes by whether they were eventually deleted or whether they remained in the code for the duration of the task. Across all 13 participants, of the 32 pasted blocks that were only partial examples, 18 (56%) were eventually deleted. Of the 26 pasted blocks that were whole examples, 19 (73%) were eventually deleted.

Participants frequently made changes to the code blocks that were pasted in. We classified these changes into three categories: "reformat" for changes to whitespace and other non-alphanumeric characters, "modify" for changes that did involve alphanumeric characters, and "undo/redo." Figure 3.5 shows the transitions that participants made between these actions, with kept pastes on the left and deleted pastes on the right. Like in Figure 3.3, thicker arrows represent more frequent transitions.

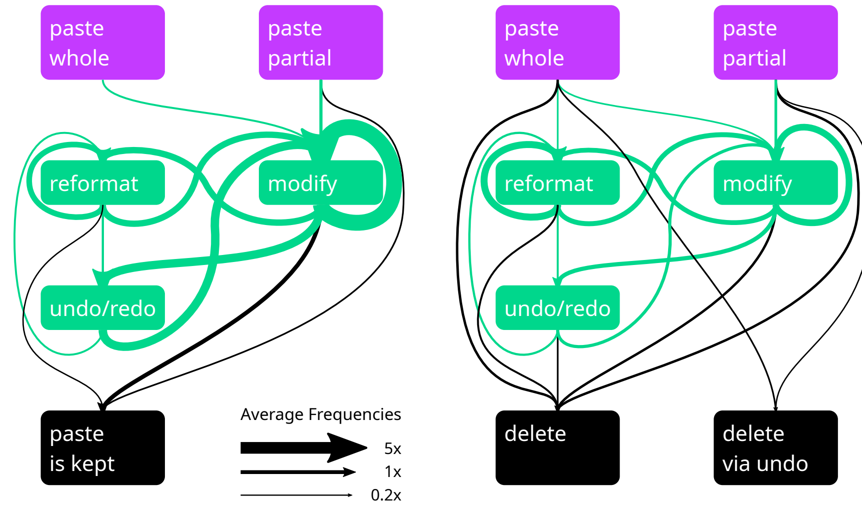


Figure 3.5: Transitions Between Edit Actions

One can see that transitioning between the various methods of editing was more common in kept pastes rather than deleted pastes; however, it was more common to make no edits at all in pastes that were deleted rather than pastes that were kept. Of the 33 pastes that were edited in some way, 16 of them (48%) were eventually deleted. In contrast, of the 25 pastes that were not edited at all, 21 of them (84%) were eventually deleted, indicating a correlation between editing a paste and keeping it.

These differences may be explained by the types of cognitive load involved. Participants who copied only part of an example or who edited an example may have been more likely to learn which parts of the example code were relevant, requiring an investment of germane load and leading to the pasted block remaining in their code. Participants who did not make this investment may have suffered from extraneous cognitive load: in this case, a code block that was not worth keeping.

Intermediate unrelated actions are not shown in Figure 3.5. For example, if a participant pasted a whole example, viewed a webpage, and then deleted the pasted block, that would count toward the "paste whole → delete" edge. Figure 3.6 shows the edits made to pasted blocks that preceded browser actions, with browser actions separated into into the localhost testing tab and other "web" tabs. As can be seen in the figure, if a participant performed an intermediate browser action immediately after pasting, then the pasted block was most often deleted, but if a participant performed a browser action immediately after modifying the code, the pasted block was most often kept. Again, this points to a benefit from increased interaction with the pasted block; participants who checked the results of their paste immediately after pasting were less likely ultimately to keep the pasted block, when compared with a participant who made modifications before checking.

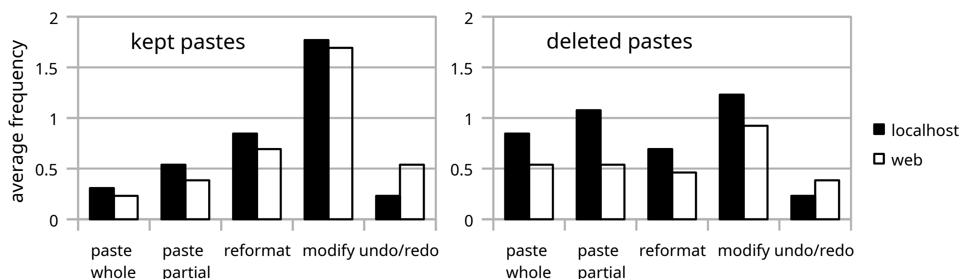


Figure 3.6: Edit Actions Preceding Browser Actions

3.2.7 Unsuccessful Pastes

As described in Section 3.2.6, 37 out of 58 copy-and-pasted code blocks (64%) were ultimately deleted. A closer look into the types of pastes that ended up getting deleted revealed some common problems. Often, problems arose from the location of the paste, rather than the code pasted itself.

One common mistake involved participants pasting code written in an "inheritance" style into their starter code, which was written in a "composition" style, creating syntax errors when methods and constructors were pasted into function bodies. Other common failed pastes included pasting HTML code into a section of JavaScript code, and pasting a code snippet without relevant imports. These issues typically caused error messages to appear

in the localhost testing tab, prompting participants to remove the pasted block rather than make the necessary changes.

In these cases, the participants' failures were largely ones of program comprehension. It seemed that participants did not invest the germane load necessary to understand what would have allowed a pasted block to function correctly, and this caused extraneous cognitive load further down the line in the form of a failed paste.

3.2.8 Cognitive Load

The NASA-TLX includes ratings of several aspects of cognitive load. Figure 3.7 plots mental effort (left) and frustration (right) for all participants at each of the twenty-minute time intervals. With some exceptions, the cognitive load scores remained mostly flat for the duration of the task, and unfortunately no clear conclusions can be drawn from this data.

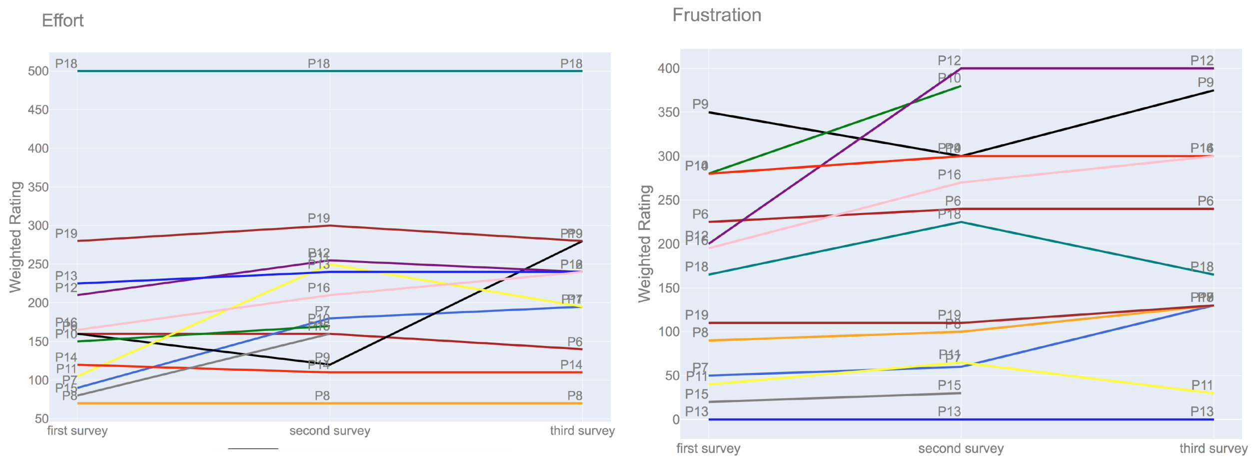


Figure 3.7: Ratings of Mental Effort and Frustration via the NASA-TLX

Chapter 4

Conclusions

4.1 Discussion

Our results do seem to validate the COIL model. For one thing, as described in Section 3.2.1, all participants performed all of the actions predicted in each stage of the model. Conversely, we noticed no action taken by participants that seemed out-of-place in the model; all actions taken by participants fit into the actions described by the COIL model.

4.1.1 Relevance of External Memory

External memory does not frequently come up in discussions of API learning. However, it is a vital component of the COIL model, and our results in Sections 3.2.4 and 3.2.5 show that tabs are used extensively as external memory resources, another point of validation for the COIL model.

Somewhat unexpectedly, browser tabs seemed to be the key external memory context for our participants. We expected participants to make use of code comments to store potentially useful code snippets, but this behavior was rare. Instead, as shown in Figure 3.5, deleting unwanted code was common. However, this effect may partially be a result of the fact that React files incorporate both HTML and JavaScript syntax, so there may have been uncertainty around comment syntax and around the relevant keyboard shortcuts in Atom. Results here may be different when developers are using a more familiar programming language and development environment.

4.1.2 Sources of Cognitive Load

While external memory was used heavily by participants, our results also point to inefficiencies in this use. The "flipping-through" of tabs noted in Section 3.2.5 suggests that participants are using extraneous load when retrieving information from external memory. Additionally, results from Section 3.2.4 show that participants end up returning to previously-viewed pages from search result pages fairly frequently. This behavior points to a different source of extraneous cognitive load: participants had to re-find information that they had previously seen in a potentially costly way.

Some of the results may indicate that an investment of germane load early in the task leads to a decrease in extraneous load later in the task. When pasting in blocks of code, some participants seemed to put in extra effort to understand the code that they were pasting in, either by copying in only part of an example or by modifying the pasted block before testing it. These behaviors may be a result of a sort of self-explanation, pointing to germane cognitive load, and when participants behaved this way the pasted block was more likely to remain in their code. Participants who interacted less with their pasted blocks were more likely to later delete the pasted block, indicating extraneous cognitive load.

Because the cognitive load plots in Figure 3.7 indicate that cognitive load was roughly constant for the duration of the task, it is difficult to judge which events triggered different levels of cognitive load. In retrospect, the three-interval approach to the NASA-TLX was probably not the best measurement of cognitive load, as cognitive load may have varied more on a shorter timeframe. An alternative hypothesis is that cognitive load truly was fairly constant for the duration of the task, possibly because participants were regulating their strategies to maintain a constant level of cognitive load. Further research is needed on this front.

4.2 Future Work

4.2.1 Expanded Study

This exploratory study highlighted some key features about the API learning process. However, this is only data from thirteen participants, with a particular API. It would be useful to conduct a similar study with a different API to determine to what degree our results generalize.

4.2.2 Measures of Cognitive Load

Future work on this model should aim to better understand the role of cognitive load in the API learning process. The three-stage NASA-TLX procedure ultimately did not prove very useful in narrowing down the key sources of cognitive load, and conclusions drawn regarding cognitive load were mostly inferred from participant behavior.

Pupillometry is a promising method to indirectly measure cognitive load. This technique measures pupil size, which has been shown to be a good measure of cognitive load in other scenarios, such as driving simulations [25]. This technique has several advantages over the NASA-TLX; for example, it can be done without interrupting the participant, and it allows for measurements over smaller time scales (several seconds), which would result in far more measurements over the course of the task. This would allow for the testing of hypotheses about which actions have a high cognitive load cost and how much cognitive load varies over the course of an API learning task.

4.2.3 External Memory Management Tools

An additional area deserving of more research is the development of tools to assist with external memory management. Our results show that the way that participants use tabs to store information is likely not the most efficient system. There is an opportunity here for new tools to be developed with the COIL model in mind, which ideally would allow users

to efficiently retrieve the information they're looking for, without having to flip through multiple tabs or perform a new search query.

References

[1] Atom. https://atom.io/.

[2] Chrome APIs - Google Chrome. https://developer.chrome.com/extensions/api_index.

[3] Chromium - the Chromium projects. https://www.chromium.org/Home.

[4] Dashiki: Simple request breakdowns. https://analytics.wikimedia.org/dashboards/browsers/#desktop-site-by-browser/browser-family-timeseries.

[5] Find and replace package. https://github.com/atom/find-and-replace.

[6] NASA TLX: Task load index. https://humansystems.arc.nasa.gov/groups/TLX/.

[7] Stack Overflow developer survey 2019. https://insights.stackoverflow.com/survey/2019#technology-development-environments-and-tools-web-developers.

[8] Tutorial: Intro to react. https://reactjs.org/tutorial/tutorial.html#setup-option-2-local-development-environment.

[9] Programmable Web: API directory. https://www.programmableweb.com/category/all/apis, 2018.

[10] Atkinson, R. K., Renkl, A., and Merrill, M. M. Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. *Journal of Educational Psychology 95*, 4 (2003), 774.

[11] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2009), ACM, pp. 1589–1598.

[12] Brandt, J., Guo, P. J., Lewenstein, J., and Klemmer, S. R. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering* (2008), ACM, pp. 1–5.

[13] Chi, E. H., Pirolli, P., Chen, K., and Pitkow, J. Using information scent to model user information needs and actions and the web. In *Proceedings of the SIGCHI*

Conference on Human Factors in Computing Systems (New York, NY, USA, 2001), CHI '01, Association for Computing Machinery, p. 490–497.

[14] Duala-Ekoko, E., and Robillard, M. P. Asking and answering questions about unfamiliar APIs: An exploratory study. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 266–276.

[15] Facebook. Create react app. `https://github.com/facebook/create-react-app`.

[16] Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM) 22*, 2 (2013), 14.

[17] Horvath, A., Grover, S., Dong, S., Zhou, E., Voichick, F., Kery, M. B., Shinju, S., Nam, D., Nagy, M., and Myers, B. The long tail: Understanding the discoverability of api functionality. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2019), IEEE, pp. 157–161.

[18] Horvath, A., Nagy, M., Voichick, F., Kery, M. B., and Myers, B. A. Methods for investigating mental models for learners of apis. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2019), CHI EA '19, Association for Computing Machinery.

[19] Intons-Peterson, M. J. External memory aids and their relation to memory. In *Cognitive psychology applied*. Psychology Press, 2014, pp. 145–168.

[20] Kelleher, C., and Ichinco, M. Towards a model of API learning. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2019), IEEE, pp. 163–168.

[21] Lawrance, J., Bellamy, R., and Burnett, M. Scents in programs: Does information foraging theory apply to program maintenance? In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on* (2007), IEEE, pp. 15–22.

[22] Maalej, W., and Robillard, M. P. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering 39*, 9 (2013), 1264–1282.

[23] Myers, B. A., and Stylos, J. Improving API usability. *Communications of the ACM 59*, 6 (2016), 62–69.

[24] Niu, N., Mahmoud, A., and Bradshaw, G. Information foraging as a foundation for code navigation (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 816–819.

[25] Palinko, O., Kun, A. L., Shyrokov, A., and Heeman, P. Estimating cognitive load using remote eye tracking in a driving simulator. In *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications* (New York, NY, USA, 2010), ETRA '10, Association for Computing Machinery, p. 141–144.

[26] Piccioni, M., Furia, C. A., and Meyer, B. An empirical study of API usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on* (2013), IEEE, pp. 5–14.

[27] Pirolli, P., and Card, S. Information foraging in information access environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (1995), ACM Press/Addison-Wesley Publishing Co., pp. 51–58.

[28] Robillard, M. P., and Deline, R. A field study of API learning obstacles. *Empirical Software Engineering 16*, 6 (2011), 703–732.

[29] Van Merrienboer, J. J., and Sweller, J. Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review 17*, 2 (2005), 147–177.

Web Resources in API Learning, Voichick, M.S. 2020