A Theory of Robust API Knowledge

KYLE THAYER, The Information School, University of Washington, Seattle
SARAH E. CHASINS, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
AMY J. KO, The Information School, University of Washington, Seattle

Creating modern software inevitably requires using application programming interfaces (APIs). While software developers can sometimes use APIs by simply copying and pasting code examples, a lack of robust knowledge of how an API works can lead to defects, complicate software maintenance, and limit what someone can express with an API. Prior work has uncovered the many ways that API documentation fails to be helpful, though rarely describes precisely why. We present a theory of robust API knowledge that attempts to explain why, arguing that effective understanding and use of APIs depends on three components of knowledge: 1) the domain concepts the API models along with terminology, 2) the usage patterns of APIs along with rationale, and 3) facts about an API's execution to support reasoning about its runtime behavior. We derive five hypotheses from this theory and present a study to test them. Our study investigated the effect of having access to these components of knowledge, finding that while learners requested these three components of knowledge when they were not available, whether the knowledge helped the learner use or understand the API depended on the tasks and likely the relevance and quality of the specific information provided. The theory and our evidence in support of its claims have implications for what content API documentation, tutorials, and instruction should contain and the importance of giving the right information at the right time, as well as what information API tools should compute, and even how APIs should be designed. Future work is necessary to both further test and refine the theory, as well as exploit its ideas for better instructional design.

CCS Concepts: • **Social and professional topics → Computing education**; *Employment issues*; Adult education;

Additional Key Words and Phrases: Software engineering, API documentation, computing education

ACM Reference Format:
Kyle Thayer, Sarah E. Chasins, and Amy J. Ko. 2020. A Theory of Robust API Knowledge. *ACM Trans. Comput. Educ.* 1, 1 (December 2020), 41 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As industry continues to expand, developers have created numerous application programming interfaces (APIs), including libraries, frameworks, web services, and other reusable code. For example, in the popular Node Package Manager (NPM), there are more than a half a million APIs available [1]. Students learn APIs in classes or in coding bootcamps [47]; developers learn and use them to build software in their jobs [3]. These APIs are what allow developers to make

Authors' addresses: Kyle Thayer, The Information School, University of Washington, Seattle, Seattle, WA, 98195, kmthayer@uw.edu; Sarah E. Chasins, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, California, 94720, schasins@cs.berkeley.edu; Amy J. Ko, The Information School, University of Washington, Seattle, Seattle, WA, 98195, ajko@uw.edu.

1946-6226/2020/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

sophisticated software quickly by reusing the work others have done and shared, and developers consider available APIs as the most important factor when choosing a programming language [29].

When developers set out to use and learn APIs, they often find a gap between what they want to do with an API and what resources they can find. For example: official documentation is often incomplete, inadequate, or hard to navigate [13, 36, 38, 42, 49], relevant examples are difficult to find and to adapt [12, 36, 42, 52], crowdsourced documentation like Stack Overflow is often incomplete, incorrect, or misleading [35]. Without a deep knowledge of the API, they are likely to produce defective or brittle code [23, 42].

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

To provide initial empirical evidence for our theory, we ran a study that tested the role knowledge plays in understanding and task progress. We recruited students, created sets of tasks using four APIs, and controlled which components of core API knowledge students could access. We measured their progress and perceived understanding of code. In the rest of this paper, we describe our theory, study, and results, then discuss their implications on API learning and the design of API instruction.

2 LITERATURE REVIEW

2.1 What is a theory?

There are many conflicting views on what constitutes a scientific theory and how to evaluate these theories [5, 9, 10, 19, 46]. The criteria that can be used to evaluate a theory include: how well it unifies and organizes facts and other components into a coherent model, what it captures about the meaning and internal logic of a situation, what it captures about causes and effect, what situations it applies to, how well it aids in answering questions in the field, what further questions the theory suggests, how well it fits current findings, whether the theory makes falsifiable hypotheses that can be tested, and whether those hypotheses hold up under experimentation [5, 10, 19, 26]. Which criteria a theory should be evaluated on depends on the particular purpose and use of the theory [5]. For example, falsifiability may be at odds with the purpose and use of some theories [5], but for theories that do produce falsifiable claims, an accumulation of contrary evidence may lead scientist to explore new avenues, potentially modifying the theory, or abandoning it entirely [19].

[1]In developing our theory, we chose to take a cognitivist perspective [8] rather than a sociocultural one. Though this ignores important factors like identity factors, social factors, communication factors, and developmental factors, it does allow us to focus on the content of knowledge about an API and build upon prior work with a similar focus.

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

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

2.2 API knowledge

Prior work on APIs has investigated the challenges and process of using and learning APIs. Developers seek knowledge from tutorials, documentation, experts, or in-line comments in code [16, 22, 27, 34]. Developers use different strategies in learning APIs (e.g., trying to understand everything before touching code or modifying unknown code to see what it does) [6, 7, 28]. Most developers prefer the strategy of looking at examples [12, 21, 28, 33, 36, 38, 40, 41, 45, 51, 52].

When developers try these strategies, they often find a gap between what they want to do with an API and what resources they can find. There are gaps in formal [13, 36, 38, 42, 49] and informal [35] documentation. When seeking examples, developers can sometimes find examples of how to use an API that they can copy and paste into their code without needing to understand the API [42]. When an example exactly matches their needs, this method can be effective at helping developers leverage an API [4]. However, this method does not always work. If a developer cannot find an example that is exactly what they need, or they cannot determine whether a found example is relevant, or they need to modify their code later, they will need a deeper understanding to successfully adapt it. Without that knowledge, they are likely to produce defective or brittle code [23, 42].

Though previous work has not offered an overarching explanatory theory of API knowledge (at least by our definition in 2.1), there have been many discoveries that pave the way toward a theory. These studies have found that developers need to know:

- What possibilities an API allows a developer to create [28]
- Precisely what behavior they are trying to achieve [21]
- How to set up an environment in which they can use the API [28]
- An overview of the API's architecture, including its components and structure [28]
- The purpose of code they want to reuse and the rationale for its pieces [2, 22, 36]
- How to search for and recognize relevant information; this involves knowing search terms and concepts [13, 20, 21, 24, 52]
- How the abstractions of an API map to abstractions used in the domain of the API [2, 6, 17, 25]
- How to use API features (e.g., object construction) [2, 12, 21, 30]
- Dependencies and relations between parts of APIs or different APIs [11–13, 20, 21, 30]
- Common or useful usage patterns and best practices [17, 20, 38, 52]
- Understanding the relationship between code and output [21, 22]
- How to find run-time information and debug programs [21]
- Runtime contracts, side effects, return values, and other properties of API behavior [11, 16, 25]

Additionally, researchers focusing on general software engineering have emphasized:

- *Programming plans*, which "represent stereotypic action sequences in programming" such as the use of algorithms or data structures [43]
- *Domain plans*, which are sequences of actions considered at the domain level, separate from the code and low-level algorithms. [50]

3 THEORY OF ROBUST API KNOWLEDGE

Effectively understanding and using an API requires a robust knowledge of the API, which consists of: domain concepts, execution facts, and API usage patterns.

3.1 Three Components of Robust API Knowledge

Building upon prior work, our own experiences learning APIs, observing developers use APIs across many studies, and teaching APIs in classes, we created a theory of robust API knowledge that distills much of evidence into a simple, explanatory theory of what API knowledge is and how it structures API learning. Our theory begins from the premise that *using* an API, much like any programming, involves imagining requirements for program behavior, and then searching for a program that meets those requirements. This process requires learning three classes of knowledge (Table 1) : *domain concepts* that exist in the world outside the API which the API attempts to model, along with the terminology used by the API; *execution facts* about the API's runtime behavior that summarize that behavior into rules about inputs, outputs, and side-effects; and *API usage patterns*, which are modifiable code patterns, along with rationale for why the pattern works and is organized as it is. We argue that these three components of knowledge, when present, allow a developer to productively find relevant abstractions in an API and correctly leverage the API's behavior to compose API abstractions into programs that meet their goals. We therefore refer to the set of these three components of knowledge as *robust* API knowledge.

In the rest of this section, we describe and justify these three components of API knowledge in more detail including their interactions (see Fig. 1), whether the knowledge our theory describes is necessary, sufficient, and how our theory meets criteria for falsifiability, explanation, prediction, and consistency with prior evidence.

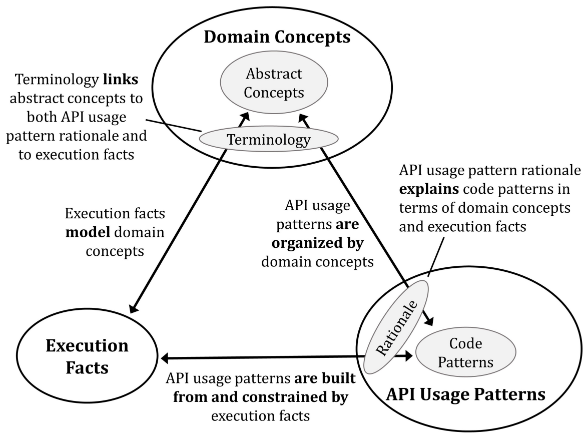

Domain Concepts
Abstract Concepts
Terminology
Terminology **links** abstract concepts to both API usage pattern rationale and to execution facts
Execution facts **model** domain concepts
API usage patterns **are organized by** domain concepts
API usage pattern rationale **explains** code patterns in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

Fig. 1. Relationships between the three components of API knowledge.

3.1.1 *Domain Concepts.*

Domain concepts are abstract ideas that exist outside of an API, which an API attempts to model, and the specific terminology that the API and documentation use to refer to the concept (see Fig. 1). These concepts are any idea modeled in software. For example, this paper is written in LaTeX, which models concepts from the domains of typography and technical writing like fonts, glyphs, tables, and

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

Table 1. Summary of the three components of robust API knowledge.

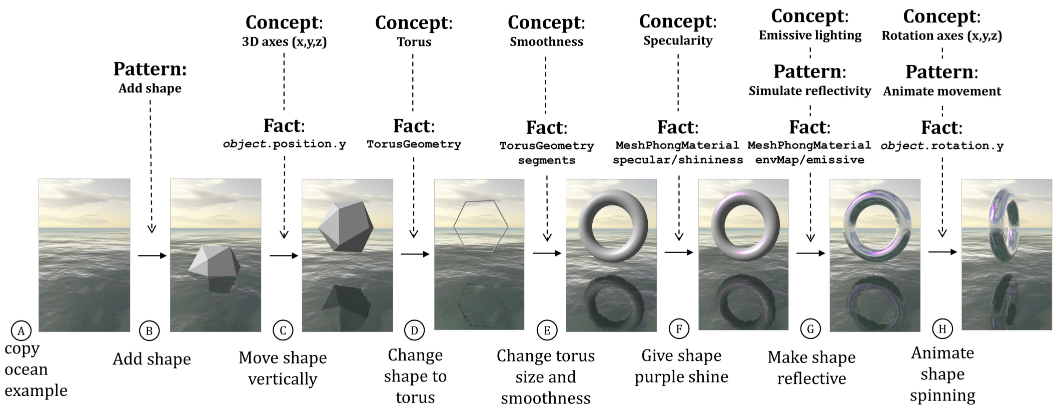


Fig. 2. The stages a developer might go through to create a spinning 3D ring floating over an ocean using the ThreeJS API. We propose knowledge needed to perform each step (further explained in section 3.1). In our study, this was one of the four task sets given to participants.

cross-references. These concepts exist in the world, outside of LaTeX, and are modeled in LaTeX. Our theory argues that someone learning the LaTeX APIs needs to understand both the concepts that LaTeX models from typography and technical writing, but also how those concepts are referred to within LaTeX. For example, to understand what the command `textit{}` does, one needs to understand the concept of italics, and that the "it" in `textit{}` refers to the concept of italics. APIs might include concepts from multiple domains, so an API might calculate screen layouts (UI design domain) by using constraint solvers (computing domain).

Consider another example, this time from an API called *ThreeJS* (which we later use in our study), which is used to create 3D animations in web browsers. Figure 2 shows several iterations of a loading screen a developer might want to create, consisting of a reflective 3D ring spinning above an ocean. In this imagined design exploration scenario, a developer must have knowledge of numerous concepts to succeed at each step. For example:

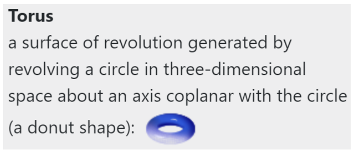

Torus
a surface of revolution generated by revolving a circle in three-dimensional space about an axis coplanar with the circle (a donut shape):

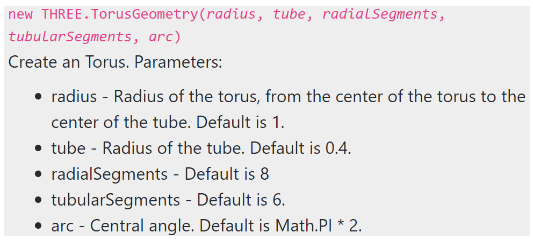
Fig. 3. Concept definition of a Torus. Used in our study.

- To move a 3D shape vertically (stage C), a developer needs to understand concepts of *3D axes* and the names of these axes in ThreeJS (x, y, and z).
- To make a ring (stage D), a developer needs to know that the ring shape they are seeking is called a *Torus*. Figure 3 shows how the concept of a torus might be explained.
- To give their shape a purple shine (stage F), a developer needs to know that this is called a *specular highlight*, how specular highlights behave in 3D scenes with lighting, and how ThreeJS refers to specular highlights.
- To make the shape reflective (stage G), the developer need to know that *emissive lighting* makes a shape a uniform color unaffected by light (so that later they can make the reflection and the specular highlight the sole determinants of the shape's color).
- To make their shape spin (stage H), they need to know what *rotation axes* are, how ThreeJS refers to them, and, in particular, that rotation around the y axis is their goal.

Broadly, our theory argues that domain concepts help developers in a number of specific ways. Concepts help developers consider what may be possible in an API, manipulate API abstractions that align with those conceptual abstractions (as mentioned in prior work [6]), and help a developer understand the purposes of API abstractions and code using the API (see Table 1). Additionally, knowing the concepts and terminology helps developers find and recognize relevant information about an API, both of which prior work has shown are critical to API learning [12, 13, 20, 24, 28, 52].

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

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


```
new THREE.TorusGeometry(radius, tube, radialSegments,
tubularSegments, arc)
```
Create an Torus. Parameters:

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

Fig. 4. Fact for the TorusGeometry function in Threejs. Note that by default, tubularSegments is 6, making the default shape a hexagon. Used in our study.

3.1.2 Execution Facts. Execution facts are declarative knowledge in the form of simplified rules about an API's underlying execution behavior, sufficient for predicting, understanding, and explaining API execution. Each of these facts models some set of concepts in terms of programming constructs such as types, inputs, outputs, and side effects of executing parts of an API (see Fig. 1). These facts can be at different levels of abstraction, from low-level information like the effect of function arguments on function return values, to higher-level information about the APIs internal state and the control and data flow of an API's global behavior [25]. Execution facts also include how an API might failure to model the domain concepts the API claims to model (i.e. implementation bugs) and how the API's behavior depends on the execution environment (e.g., 'this code will not work in Internet Explorer').

For example, completing our ThreeJS scenario (Figure 2) might require knowing these facts:

- To move an object vertically (stage C), a developer needs to know that the property `object.position.y` changes the object position.
- To create a Torus object (stage D), a developer needs to know that the `TorusGeometry` constructor creates a Torus which can be rendered when added to a scene.
- To create a smoothed Torus with the target size (stage E), a developer needs to know facts about the effect of the first four parameters of `TorusGeometry` (Figure 4). In particular, they need to know that the first two (`radius` and `tube`) define sizes and the second two (`radialSegments` and `tubularSegments`) define geometric smoothness.
- To give the object a purple shine (stage F), a developer needs to know the behavior of options for `MeshPhongMaterial`, in particular that `shininess` defines how sharp a specular highlight is and `specular` defines the color of the specular highlight.
- To make an object appear reflective while maintaining the purple specular highlight (stage G), a developer needs to know that `MeshBasicMaterial` does not allow specular highlights and that they must use `MeshPhongMaterial` instead, setting the `emissive` color to white to make it uniformly lit and turning off the main `color` by setting it to black. They still need to leave the `specular` and `shininess` values to keep the purple shine.

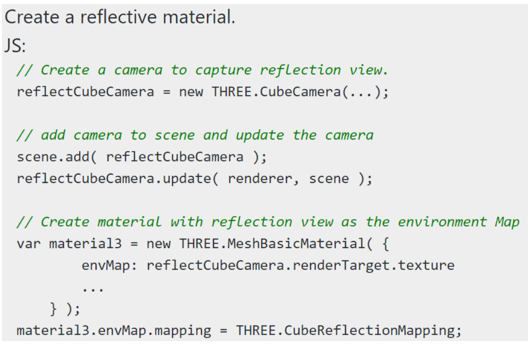
- To animate the object (stage H), the developer needs to know that the property `object.rotation.y` changes an object's rotation.
- For all steps they need to know that changes will be visible on the next frame rendered.

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

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

The ability to reason about and predict API execution behavior is central to many software engineering activities. It helps developers efficiently test, debug and repair code [21, 38], and interpret error messages or other unexpected behavior. Additionally, being able to predict API behavior helps developers judge between options when modifying code or writing new code.

3.1.3 API Usage Patterns. API usage patterns are some form of a code pattern (e.g., code examples, ordered lists of API calls) that conveys how parts of it may be modified. Given how often multiple APIs are used together, these patterns might include the use of multiple APIs in coordination. We additionally consider API usage patterns to include rationale (whether explicit or implicit) for pattern's construction in terms of both the concepts that the code is organized around (e.g., the code may implement a specific algorithm, heuristic, trick, or convention known in the domain) as well as how the execution facts of the API work together (or must be worked around) to produce the desired result (see Fig. 1). [2]


Create a reflective material.
JS:

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

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

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

Fig. 5. API usage pattern for creating a reflective material in Threejs. Used in our study.

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

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

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

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

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

[2]Various definitions of API "usage patterns" have been used before (see the Robillard et. al.'s literature review [37]). These have generally been defined concretely in terms of ordered or unordered sets of API calls, and sometimes with additional information like control structures [32]. Our definition here is more general, and unlike those definitions, we include rationale.

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

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

Consider other examples from our ThreeJS scenario (Figure 2):

- To add a shape to a scene (stage B), developers must first create a Geometry and a Material, then create a Mesh object from those, and then add the Mesh object to the scene. An API usage pattern for this process would show: (i) the order in which to create objects for a Mesh, (ii) that ThreeJS can be used for adding shapes to scenes, and (iii) where to change the code to add different shapes to a scene.
- To make an object appear reflective (stage G), developers must create a `CubeCamera`, add it to the scene, update it (to render), then set the material of the reflective object to use the camera as an environment map with `CubeReflectionMapping`. For example, Figure 5 shows an API usage pattern which shows (i) the sequence of function calls for completing this task, (ii) that ThreeJS objects can be reflective, and (iii) where to change the settings of the `CubeCamera`.
- To animate an object, a developer must store it in a variable with a high scope, then in the callback argument of `requestAnimationFrame` (a function provided by JavaScript), they can get the current time and set the position or rotation of the object. An API usage pattern for this task would indicate: (i) that animating ThreeJS objects is possible, (ii) where to put code for animation and how the code should access the object to animate, and (iii) where and how to change the code to change how the object animates.

Prior work shows that code examples are essential to API learning, whether pseudocode that foregrounds usage rules and patterns or full executable examples [12, 21, 28, 33, 36, 38, 40, 41, 45, 51, 52]). However, evidence also shows that it is essential for API usage patterns to explain usage rules and their rationale [13] and why the API was designed in the way it was [22, 28, 31, 38]. Prior work also shows that API usage patterns are key to revealing the range of behaviors that an API makes possible. Examples define a single point in a design space, while an API usage pattern shows a range of possibilities. A developer must have a rich mapping between the behaviors that an API enables and the code that implements them in order to overcome selection barriers [21, 38].

3.2 Making Use of Robust API Knowledge

Now that we've defined the three components of robust API knowledge (see Table 1) and defined how they relate to each other (see Fig. 1), we can consider how a developer makes use of this knowledge together when working APIs. First, we will consider the perspective from which this knowledge is central, and then we will consider what happens when a piece of knowledge is missing.

3.2.1 Where this knowledge is central. We claim that from the limited perspective of an isolated developer understanding code that uses an API, considering options for using an API, and writing or modifying code that uses an API to achieve a specific behavior, concepts, facts, and patterns are central for a developer. These are not the only central components of knowledge needed though, since, for example, a developer also needs to know the underlying programming language (which has its own concepts, facts, and patterns). Many other types of relevant API knowledge (e.g., location of documentation, IDE auto-complete functions, code generators) is secondary from this perspective, aiding in learning and interacting with concepts, facts, and patterns.

Still, creating code in isolation is not the only perspective worth considering when working with APIs, and in many other situations, concepts, facts, and patterns are not sufficient. If your goal is not just to create code that would achieve a specific goal, but also to run that code, you may need to know how to install the API and execute code, requiring knowledge of things like operating systems, platforms, and configuration issues. Code is generally written to be understood by other developers, adding readability as an important part of the rationale of API usage patterns, and requiring knowledge about how to work with and communicate with other developers. APIs also exist within their own active communities. Taking an active role in these larger communities may involve needing knowledge about community standards, where to ask questions, how the API changes over time, how to report bugs or upgrade API versions.

3.2.2 When knowledge is missing. Since a developer might only be interacting with some portion of an API for a given task, it would certainly not be necessary for developers to know the near infinite number of concepts, facts, and patterns about the whole API. So, a developer can certainly go without knowing many pieces of API knowledge. Still, there will be some relevant set of concepts, facts, and patterns that, if missing, will hamper a developer in achieving their goal. Even then, if a developer is missing knowledge of a particular concept, fact, or pattern, they still might be able to infer it from the other components of API knowledge:

- *Inferring domain concepts*: A developer might look at a fact or pattern that uses a term they hadn't known before and infer the term's conceptual meaning.
- *Inferring execution facts*: A developer might recognize terminology used in an API element name and guess at how its output relates to the known concept. Alternatively, a developer could look at how and why an API element is used in an API usage pattern and infer some of the execution rules around inputs, outputs, and side-effects of the API element.
- *Inferring usage patterns*: A developer could take a domain concept they know around how to achieve some goal, and then use their knowledge of execution facts of the API to piece together and mentally check potential patterns of code until they successfully invent an API usage pattern they had not known before.

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

3.3 Evaluating our theory

Earlier we made three claims for what a successful theory of API knowledge would do (2.1), so here we evaluate our theory by those three criteria.

3.3.1 Unify prior findings on API knowledge into a coherent model. Our theory of API knowledge is not based on a single source of evidence, but on a diverse collection of prior work, not solely conducted by the authors. Our theory incorporates many previous findings: Domain concepts help developers seek and recognize relevant information [13, 21, 24, 52], understand the purpose of code [22, 36], and decide what purpose they want new code to fulfill [21, 50]. API usage patterns demonstrate what is possible with an API [28], help developers understand how API code works together and how to coordinate API use [21, 22, 36], and encode common usage patterns and best practices [38, 43, 52]. Execution facts elucidate the details of the API structure [28], how to use API features [12, 21, 30], how to understand dependencies and relations between API features [12, 13, 30], and details of how code relates to output [21, 22]. We did not incorporate previous findings that cover API-related knowledge we explicitly exclude from our theory: initial setup of an API that involves the environment [28], how to find information about the internal and external behavior of code [21], and programmers' misapplication of knowledge [23].

3.3.2 Capture the internal logic of how parts of API knowledge relate to each other and the role API knowledge plays in helping a programmer work with code that involves APIs. Our theory attempts to explain why concepts, facts, and patterns are central components in robust API knowledge and how those components relate to each other. We argue that in the (admittedly limited) context of an individual reading and writing (but not running) API code, concepts, facts, and patterns are some of the core components of knowledge that the developer needs. We explain the role of each component of knowledge in understanding and designing code (Table 1), and how each component relates and connects to the others (Fig. 1). Together these capture how it is that a developer knows (or fails to know) what to search for, what it means, how to adapt it, and how to fix it when it breaks.

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

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

In the rest of this paper we will present a single study as a start, testing five hypotheses generated from our theory. In the study, we investigate the impact of access to concepts, facts, and patterns on productivity and comprehension. This study is just a start to validating our theory through empirical studies. More research will be necessary to further test, refine, or refute the theory's claims.

4 STUDY: THE IMPACT OF ROBUST API KNOWLEDGE

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

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

- **H.2.a** Learners' self-reported comprehension of the API will increase when they have access to each of the three components of API knowledge.
- **H.2.b** Learners' self-reported comprehension of the API will increase when participants have access to a larger number of components of knowledge.
- **H.3** Participants will report value in concepts, facts, and patterns for tasks involving an API.

To test these five hypotheses, we designed a controlled laboratory experiment that presented a series of web development task requiring the learning and use of several different APIs in a short period of time. We recruited university students and varied participants' access to each of the three components of knowledge in our theory. This design allowed us to begin to understand the causal impact of different components of knowledge on both task progress and self-reported comprehension.

4.1 Method

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

4.1.2 Tasks. In designing tasks, we sought diverse, ecologically valid uses of APIs that reflected a developer trying to build something with an unfamiliar API. We chose four different JavaScript APIs, attempting to cover different domains of use, as well as APIs with concepts we believed our participants would likely not know. We chose the four APIs and created the following task sets for each one (see Appendix A.1, A.3, and the supplemental materials[3] for more details):

- **d3.js** (d3js.org) is a data visualization API that requires knowledge of scalable vector graphics (SVG) and concepts related to data visualization such as marks, color theory, and data mappings. The d3.js task set asked participants to improve a visualization of known exoplanets.
- **Natural** (github.com/NaturalNode/natural) is a small natural language processing and information retrieval API. It requires knowledge of linguistic concepts such as syntax, stemming, and parsing. The Natural task set asked participants to add code to a stubbed-in interface, so the program would process the text of Jane Austin novels and users could see words associated with characters and search for characters by word.
- **OpenLayers** (openlayers.org) is an interactive mapping API. It requires knowledge of concepts in geographic information systems such as projection, markers, and layers. The OpenLayers task set asked participants to improve an interactive map by making it be rounded, with latitude and longitude lines, country outlines, and allow users to draw on the map.
- **ThreeJS** (threejs.org), the 3D graphics library we described in the section 3, requires detailed knowledge of geometry, physics, and lighting. The ThreeJS task set asked participants to create a loading screen by adding a spinning, shiny ring to an ocean scene.

For each task set, we gave participants several materials and forbade them for looking for additional resources online. We created starter code and a development environment, which eliminated the need to learn how to install and configure the API. We provided a description of each

[3]Supplemental materials also available at https://github.com/kylethayer/API-Knowledge-Theory-Supplemental-Materials

task in the task set that they were to achieve, along with instructions that their goal should not just be to solve the tasks, but to understand how the code works. Finally, we provided a code example from which they could learn the API, mimicking the discovery of an example in documentation, a blog post, or a StackOverflow answer.

The starter code for each task set ranged from a 30 line JavaScript program to make a simple map for OpenLayers to a 204 line JavaScript program to make an interface to show the results of using the Natural library. We based most of the starter code on examples that were similar to or exactly matched the examples provided in each API's documentation, but with comments stripped out, so we could isolate the effects of the information we provided on concepts, facts, and patterns.[4]

The example code we provided was designed to be fairly close to what the participants needed to complete the tasks, including at least one call to every API function needed to solve the tasks, but were different enough that participants could not directly copy and paste code to solve the tasks. They had to change at least some parameters of function calls to achieve correct behavior. This principle of "near match" mirrored what learners are likely to find online on sites like StackOverflow, but also made the tasks feasible to complete in the short 15 minute per API time limits.[5] We also designed example code to minimize the dependence on knowledge of JavaScript, primarily relying on conditional logic, function calls, and simple object literal declarations. This ensured that that participants' progress would depend primarily on understanding of the API, and not on potentially brittle knowledge of JavaScript's language semantics.

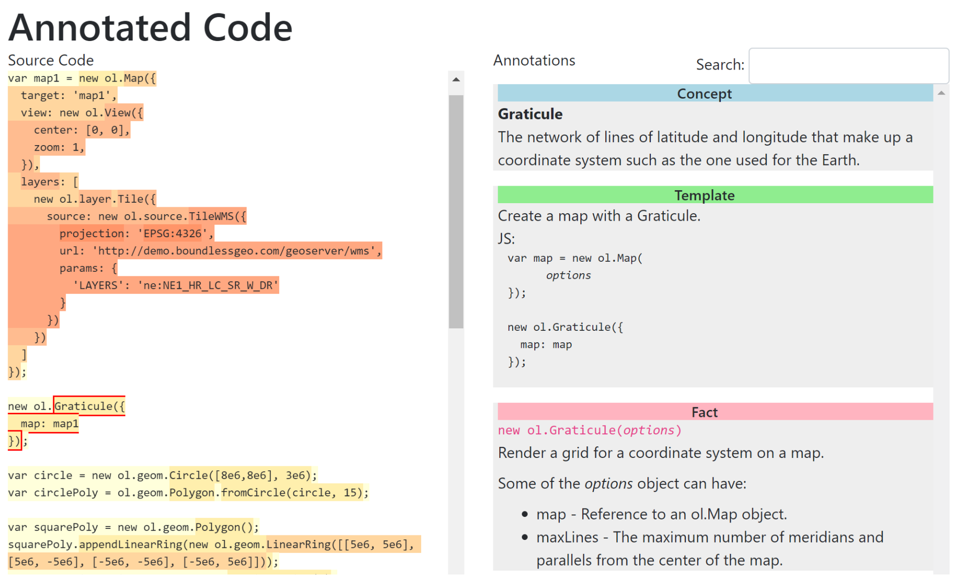
We piloted each task with multiple participants with sufficient prior knowledge, clarifying unclear instructions, adding missing annotations, and adapting task difficulty.

4.1.3 Manipulating access to API knowledge. The core manipulation of our experiment was varying access to our theory's three components of API knowledge. To do this, we formatted the examples that participants received with *annotations* on each line of code with the three components of API knowledge. For example, when the example code referenced a *concept* that we believed many participants would not know (e.g., pack graph, torus), we wrote a definition, generally based on Wikipedia or Wiktionary (Figure 3 shows an example of one of these definitions). For *API usage patterns*, we looked for multiple lines of code that we judged to work together to achieve some purpose, and created a commented code template that represented the usage pattern (Figure 5 shows an example template).[6] For each line of example code calling a function, we created function descriptions to teach execution facts about the function's behavior (Figure 4 shows an example description of a function's execution facts). We generally pulled *facts* directly from the official documentation, though we removed info on parameters not used in the code from some functions to reduce the annotation length, especially for functions that had dozens of options like ThreeJS's `MeshPhongMaterial`. In annotating the example code, we focused on explaining the whole example code, instead of limiting ourselves to just the annotations relevant to the tasks (we didn't want the annotation highlights to only be on the parts they needed, otherwise the highlights might provide hints separate from the annotations). We created these annotations ourselves and the only

[4]The main exception to this was our Natural starter code, which mostly consisted of code to load the book files and run the user interface (see Fig. 8). We commented the Natural code extensively and indicating where modifications would need to be made. This was intended to allow users to focus more on the Natural API code they would have to add to the interface and not the interface itself. Two other places comments were added were to the ThreeJS task marking "Additional code that you don't need to worry about: creating water, animating water, creating sky, responding to resize," and to OpenLayers indicating the code that created the Sphere Mollweide projection, which used another API (Proj).

[5]Since we wanted our participants to work with multiple APIs and be in multiple conditions to aid in comparisons across conditions, and we wanted it to be short to make it easier to recruit for, we ended up only having 15 minutes per API.

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



Annotated Code

Source Code

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

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

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

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

Annotations

Search:

Concept

Graticule

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

Template

Create a map with a Graticule.

JS:

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

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

Fact

`new ol.Graticule(options)`

Render a grid for a coordinate system on a map.

Some of the *options* object can have:

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

Fig. 6. Annotated example code from our study's OpenLayers task set. Code with annotations are highlighted (darker highlight for more annotations). The example code calling `Graticule` is selected and relevant annotations are displayed on the right.

validation we did for the quality of the annotations was by piloting the task sets. This piloting involved having several individuals (not included in the study) try a draft of the experiment with a researcher observing and asking about difficulties or confusions after. Based on this piloting we made some modifications and additions to the annotations to improve them. For example, after seeing pilot users express difficulty comprehending the initial definition of n-grams, we added example n-grams to the definition.

Figure 6 shows the interface we created to present the code example and its annotations.[7] The tool highlighted code that had annotations and when participants hovered over the highlights or clicked on them, the right panel showed the relevant annotations. For example, in Figure 6, the `Graticule` code is selected and annotations are shown for the concept of a Graticule, the template for creating a map with a Graticule, and the facts about the `Graticule` constructor. For times where we gave participants no annotations, none of the code was highlighted and annotation area said, "No annotations provided for this task". Our goal in allowing participants to select code and see the annotations relevant to it was to reduce the time and effort spent on information retrieval, that way we could see better the effect of the information itself. The tool allowed us to hide or show annotations by knowledge components so that we could test which type of annotations had which benefits to participants in different experimental conditions.

We tested the effect of the different knowledge components by changing which annotations were visible for which API task sets to participants.[8] Each participant had one API task set with no annotations, one with one annotation type, one with two annotation types, and one with all three annotation types. We counterbalanced which annotations they had and for which API task sets to randomize any learning affects that occurred between tasks Table 7 shows our various counterbalanced conditions, following a Latin square pattern. We assigned each participants a

[7]The annotated code examples are available at https://github.com/kylethayer/API-Knowledge-Theory-Annotated-Code.
[8]Some participants saw a mislabeled fact which we fixed for future participants. When we tried including it in our statistical models it was never a statistically significant factor, so we exclude it from all models reported in this paper.

condition number. We continued adding participants even after the initial 48 (twice for each condition) to increase the amount of data we had to work with for a total of 54 participants. While adding those extra participants, we gave some conditions where there had been a technical difficulty in some of the task sets so that we would have two complete sets of data for each task set (we still used the partial data from the people who had technical difficulties). For each participant we also assigned them a random order of API task sets and a random vertical display order of the annotation types (for when there were more than one type).

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

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

When participants arrived, we gave each a consent form and a survey including questions on how many programming languages and programming libraries they had learned, as well as a rating of their confidence in debugging JavaScript in Chrome. We then gave them instructions that described how the setup of the coding environment (including the Cloud9 IDE) and the three components of core API knowledge which they would see in annotations. We then gave them a sample task to work on using a fifth API (slickGrid) that included annotated example code, so they could get used to navigating our setup. We told them that when it was time to work on the different API task sets, to complete as many tasks as they could in order (see Appendix A.3 for survey and instructions).

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

Fig. 7. The 24 counterbalanced conditions of API and annotations available used our study.

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

4.2 Analysis and Results[9]

Below, after discussing measuring prior knowledge, we discuss the influence of annotations on task set progress (section 4.2.2) and on perceived understanding (section 4.2.3), then we discuss the attitudes that participants reported toward the annotations (section 4.2.4).

4.2.1 Measuring prior knowledge. To control for prior knowledge in our models and to assess inclusion criteria, and guided by evidence that people can somewhat reliably estimate their programming experience [14], we used three self-reported measures of programming experience: First, we asked participants to approximate the number of programming languages they had learned and

[9]Data and scripts are in the supplemental materials and at
https://github.com/kylethayer/API-Knowledge-Theory-Supplemental-Materials

select one of the following: (1, 2-4, 5-9, 10+).[10] The purpose of this question was to gauge general experience working with different types of code. We expected more diverse programming experience to lead to faster reading, understanding, and writing of code. Second, we asked participants to approximate the number of APIs they had learned as one of the following: (1, 2-4, 5-9, 10+).[11] We expected more experience learning APIs to help participants learn new APIs more quickly because they may had encountered concepts, patterns, or facts similar to ones in the study. Third, we asked participants to rate their *confidence in debugging JavaScript in the Chrome web browser* on a seven-point Likert-scale, as participants would be able to use the Chrome debugger.

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

Measuring task progress. As participants worked on their task sets, Cloud9 stored a history of their edits and saves. We had participants mark a task when they thought they had completed it, but we wanted to judge their work for ourselves. Since the tasks in the API task sets were challenging and often involved multiple changes to the code, we identified the necessary subtasks to solve each task (Fig. 8 and 9). We then rated how many of these subtasks were complete in participants' final code as well as two saves prior (in case they had just done something to break their code right before we called time, such as deleting a line of code in preparation for rewriting it) by reading and executing the code. We then recorded the *task set progress*, meaning: the most subtasks completed in any of those three versions of their code.

One researcher (Kyle) rated the progress on all tasks of all users, and then another researcher (Sarah) independently rated the progress on 10%. Our inter-rater reliability check found that 64% matched exactly. The low number was mostly due to the difficulty of rating progress in d3.js (only 38% matched), where few participants found the example code related to the first task, and came up with different solutions. Specifically, there were a number of cases in d3.js where participants solved a task in a way we had not intended (by performing task 1 by just modifying the html or using plain JavaScript instead of the d3.js library to add the title), or not finishing one task before moving on to the next. We excluded these from our d3.js task progress analysis and the researchers redid the progress rating jointly for the remaining d3.js participants. For the other three libraries, 79% progress ratings matched exactly and 93% were within one subtask. We additionally had one researcher rate completion of tasks 1.4, 1.5, and 1.7 in ThreeJS for subtask analysis and when we had another researcher do the same for 10% of participants, there was 100% agreement.

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

To build our model, we used task set progress as the dependent variable, and presence of concepts, facts, and patterns, and how many previous programming languages (PLs) learned, how many previous programming libraries (APIs) learned, and their confidence debugging JavaScript in

[10]We later realized we should have clarified whether HTML and CSS counted as programming languages.
[11]We later realized we should have included 0 in number of APIs.

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

Table 2. Ordinal regression model for task set progress in each API, testing the role of the types of annotations available to participants (H.1.a).

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

Table 3. Ordinal regression model for task set progress in each API, testing the role of the number of annotations available to participants (H.1.b).

Chrome as our independent variables.[12] Since our dependent variable (task set progress) was ordinal we used proportional odds logistic regression models.[13] We then calculate 95% confidence level p-values from the t-values and the degrees of freedom.[14]

For overall progress on task sets, certain components of knowledge helped on certain task sets, some components of knowledge hurt, and others appeared to have no effect (Table 2). In order to make sense of these apparently contradictory results about whether annotations were beneficial or not, we consider several factors and specifics in section 4.2.5. For now, we will report the results we found.

We found significant differences in the following cases: concepts decreased progress in the d3.js task set ($\beta = -2.6$, $p < .017$); patterns improved progress in the d3.js task set ($\beta = 2.1, p < .023$); patterns improved progress in the Natural task set ($\beta = 2.3, p < .007$); facts hurt progress in the Natural task set ($\beta = -2.3, p < .012$); and facts hurt progress in the OpenLayers task set ($\beta = -1.5$, $p < .027$). We also found that number of previously learned APIs had a significant positive linear correlation with task set progress in d3.js ($\beta = 2.4, p < .034$) and OpenLayers ($\beta = 2.2, p < .014$),

[12]For our ordinal variables: Learned PLs, Learned APIs, and Debug Confidence, our models fits a polynomial (the polynomial has one less degree than the number of levels), resulting in a set of coefficients for each polynomial degree: linear, quadratic, cubic, etc.

[13]Proportional odds logistic regression models fit a set of intercepts for the dependent variable (in this case task progress), one intercept for each boundary between levels of the variable. We do not report these intercepts as we are more interested in the influences from the independent variables.

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

and debugging confidence had a significant positive quartic correlation with task progress in d3.js ($\beta = 2.8, p < .028$) and a negative cubic correlation with task set progress in Natural ($\beta = -2.5$, $p < .032$).

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

We only found two instances where the number of annotations significantly influenced task set progress (Table 3): having all three significantly increased task progress with ThreeJS ($\beta = 2.5$, $p < .016$) and having one (as opposed to 0) significantly decreased progress in OpenLayers ($\beta = -1.8$, $p < .025$). The correlations with number of PLs learned, APIs learned, and debugging confidence correlated in similar ways to before.

Whereas the analysis above considered overall task set progress, some of our tasks had clear bottlenecks. We therefore chose specific subtask bottlenecks (refer to Figs. 8, 9) to investigate further. We based our choice of bottlenecks on visual examination of where it appeared that one of the annotation types had made a difference (biasing our results toward significant differences, but highlighting the impact of robust API knowledge) and also on whether the amount of data we had for those steps allowed for analysis. This resulted in five bottlenecks from three of the APIs (potential bottlenecks in d3 weren't clear enough or had insufficient data):

- **Natural N-Gram Subtask** *(completed task 1.1)*: This subtask was to start modifying how close words should be to a character's name to be considered. This involved changing a call to the *trigram()* function to a call to the *ngram()* function.
- **Natural TF-IDF Subtask** *(completed task 2.1, for those who finished task 1)*: This subtask was to start grouping text together and find words that were relatively more common in different sets. This involved using the *tfidf* functionality.
- **OpenLayers Graticule Subtask** *(completed task 2.1, for those who finished task 1)*: This task was to start adding latitude and longitude lines; this required the `Graticule` constructor.
- **ThreeJS Torus Subtask** *(completed task 1.5, for those who finished task 1.1-1.4)*: This subtask consisted of correctly choosing the *TorusGeometry* constructor to create a shape like the one in the example.
- **ThreeJS Torus Params Subtask** *(1.7, for those who finished task 1.5)*: This subtask consisted of making the Torus look smooth by putting a large number in the fourth parameter of *TorusGeometry* constructor to increase the tubularSegments.

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

Table 4 shows the resulting models for each of these bottlenecks. We found that on the Natural N-Gram subtask (1.1), patterns (which showed how various blocks of the example code worked, though there were no patterns on N-Gram use in the example code) helped the participants make task set progress ($\chi^2(1, N = 54) = 11.7, p < .0006$), while facts (which showed parameters for the n-gram methods, though didn't explain the meaning of n-grams) hurt progress there ($\chi^2(1, N = 54) = 14, p < .0002$). On the Natural TF-IDF subtask (2.1), we found that concepts (which included a definition of TF-IDF) trended (though not significantly) toward helping participants make progress ($\chi^2(1, N = 30) = 3.5, p < .063$). On the OpenLayers Graticule subtask (1.1) we

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

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

Table 4. Anova for ordinal regression of task set progress by API for specific task / subtask progress. Note that these were chosen based on appearance of annotations making difference based on visual inspection, so the p values are artificially biased toward significance.

also found that concepts (which included a definition of graticule) also helped participants make progress ($\chi^2(1, N = 45) = 8.7, p < .003$). On the ThreeJS Torus subtask (1.5), we found that concepts (which included the definition of Torus shown in Fig. 3) helped participants make progress ($\chi^2(1, N = 47) = 5.4, p < .02$). Finally, on the ThreeJS Torus Params subtask (1.7), we found that facts (which included the parameters of TorusGeometry shown in Fig. 4) helped participants make progress ($\chi^2(1, N = 36) = 30, p << .0001$), while patterns hurt progress ($\chi^2(1, N = 36) = 8.5, p < .004$). We did not find that the number of APIs previously learned had any significant influence on these specific bottlenecks.

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

Measuring Understanding. To measure how well participants learned the API at the end of their task set, we chose to measure their perceived understanding of the example code, since that code was still the same for all participants at the end of the task set and included many calls to the APIs. After each task set we gave participants a paper copy of the example code, and for each substantive line of code (we marked these for them), to rate their agreement with the statement "I understand what this line does and its purpose in the larger program."[15] on a five-point Likert scale. Not all participants finished rating the whole example code within the time. We would have liked to measure this understanding before the task set as well, but we didn't want to use the time.

[15]In our pilot studies we tried some other measurements of understanding that didn't work as well as we hoped. We tried having participants write comments on the example code, but that was slow and hard to evaluate. We also tried asking participants to separately rate both how well they thought they understood what the line does, and how well they understood the line's larger purpose in the program, but participants found it confusing and difficult to answer.

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

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

Table 5. Ordinal mixed model regression model and anova for perceived understanding in each API, testing the role of the types of annotations available to participants (H.2.a).

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

We found a number of cases where each of concepts, facts and patterns (ignoring whether the other ones were present) either increased or decreased perceived understanding in the four different API task sets (Table 5). Again, we consider several factors and specifics that may explain these apparently contradictory results in section 4.2.5. We found that concepts increased perceived understanding in ThreeJS ($\chi^2(1, N = 1927) = 23, p << .0001$), and decreased perceived understanding in Natural ($\chi^2(1, N = 1334) = 6.3, p < .01$) and OpenLayers ($\chi^2(1, N = 1850) = 13, p < .0002$). Patterns increased perceived understanding in d3.js ($\chi^2(1, N = 1391) = 9.0, p < .003$), and Natural ($\chi^2(1, N = 1334) = 32, p << .0001$), and decreased perceived understanding in OpenLayers ($\chi^2(1, N = 1850) = 15, p < .0001$). Facts increased perceived understanding in ThreeJS ($\chi^2(1, N = 1927) = 4.4, p < .04$) and decreased perceived understanding in d3.js ($\chi^2(1, N = 1391) = 5.8, p < .02$). We also found that number of previously learned programming languages always had an effect and was generally a positive one. Number of previously

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

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

Table 6. Ordinal mixed model regression model and anova for perceived understanding in each API, testing the role of the number of annotations available to participants (H.2.b).

learned APIs, and debugging confidence were also always significant, but whether it correlated with more or less perceived understanding was inconsistent.

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

We found that having more than zero annotations often improved understanding, but understanding wasn't necessarily improved when there were more than one type present (Table 6). For all four models, the number of annotations was significant: d3.js ($\chi^2(3, N = 1391) = 61, p << .0001$), Natural ($\chi^2(3, N = 1334) = 38, p << .0001$), OpenLayers ($\chi^2(3, N = 1850) = 52, p << .0001$), and ThreeJS ($\chi^2(3, N = 1927) = 18, p < .0004$). In d3.js, having one or two annotation types showed an improvement in understanding, but having three annotation types decreased understanding. In Natural, having one, two, or three annotation types all showed improvement in understanding, but having three showed the least improvement. In OpenLayers, having one, two, or three annotation types showed decreased understanding, and more so with each additional annotation type. In ThreeJS, having one, two, or three annotation types all showed improvement in understanding, and more so with each additional annotation type. The fact that we saw relatively smaller coefficients for having three annotation types with three of the APIs, points to the possibility that at some point information became overwhelming for our participants, as we'll see below in 4.2.4.

4.2.4 *Developers' Attitudes Toward Annotations (H3).*

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

One researcher started with codes for the three components of knowledge from our theory, and then did open coding to produce codes for additional knowledge and strategies, and how students valued those (code book in A.2). We combined codes for the answers to the two questions to produce a set of codes for each participant/API pair. Once we finished coding, we had another researcher independently coded 10% according to the code book, and we found that 78% of codes matched.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Seventeen API/participant pairs mentioned copying-and-pasting code from the example. For example, P13 in OpenLayers said, '*The first task was essentially copy & paste. Without understanding why it work or what it was doing in depth.*" In contrast, P24 mentioned modifying the code that was copied-and-pasted while working on ThreeJS task 1, *"Copying the example code to my JS program, which I knew would produce some specific shape, and then playing around with parameters for shape and material construction to try and get my desired result."*
- Participants were sometimes able to infer information from the example code. For example, four participants compared code from the two example maps in OpenLayers, which showed different input construction (implying API usage patterns), and the effects of those inputs (implying facts). Additionally, there were 12 mentions of variable or function names being useful. These may have helped participants infer concepts or facts about the API, such as P21 in d3.js, saying, "*Some names in the source code were helpful to make educated guesses about how I might complete a task (i.e. "r" probably having something to do with a radius).*" There were also six mentions of wanting better variable or function names, such as P51 in d3.js saying, "*Info regarding poorly-named vars ('g' in particular) would have been nice as well.*"[16]
- Some participants mentioned wanting more from examples (35 participants/API pairs of 214), such as P32 asking for another example map in OpenLayers, P20 wanting example code closer to the task, or P69 wanting to modify and test the example code in d3.js and ThreeJS.
- Two participants said example code with no annotations and a brittle understanding was limiting. P61 in ThreeJS, said, "*Comparing to the example code was useful for duplicating exactly what it did. I found it very difficult to extend the example code to something new [emphasis original].*" and P19 in ThreeJS said they did not "*know how to do anything outside of the given example. Could only replicate shapes and change values, not make new shapes.*"

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

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

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

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

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

Table 7. Summary of regression results for annotation types in our study.

4.2.5 Discussion. Table 7 summarizes the results of our regression models for annotation types from our study. Overall we found support for our hypotheses, with several exceptions:

- H.1.a: For many tasks, access to information about concepts, facts, and patterns was associated with more progress, but for others, it was associated with less.
- H.1.b: Annotations only showed a cumulative positive effect in one API (ThreeJS) where more annotation types meant more progress.
- H.2.a: For many tasks, access to information about concepts, facts, and patterns was associated with increased perceived understanding of the API, but for others, it was associated with less perceived understanding.
- H.2.b: There were mixed results on whether there was a cumulative effect: one API (ThreeJS) had improvements with each additional annotation type present, OpenLayers had detriments with each additional annotation type. The other two showed reduced benefits when all three were present.
- H3: Participants' sentiments toward the annotations were largely positive, and when participants lacked a particular component of knowledge, many expressed wanting it. However, they also indicated not understanding some annotations, and when having multiple annotations, feeling overwhelmed trying to understand them during the short 15-minute tasks.

When trying to interpret these results, particularly where annotations reduced progress or perceived knowledge, we are aware of a number of limitations with our study design. For example, the only control on the quality of the annotations was feedback from the pilot experiment. We also provided more annotations than participants needed (so the highlights showing where annotations were didn't act as a separate clue), but since we didn't measure what participants looked at, we don't know how much attention participants paid to irrelevant annotations. Additionally, which annotations were relevant changed with each subtask participants worked on, though the interface stayed the same. It is therefore likely that some participants might have even spent time reading an annotation that would have been useful on a later subtask that the participant never got to. The annotations themselves could be long and it might only be one detail that was relevant to the task. We also observed several factors with specific API task sets that help interpret these findings:

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

bunch of calls to the API (taken from the official documentation), and we think the patterns may have helped participants to more quickly focus on code segments.
- *OpenLayers*: The example code consisted of two short examples next to each other which showed all the features needed. This seemed to be sufficient for inferring many concepts, facts, and patterns, and so other than a definition of "Graticule" for Task 2.1, additional information just slowed people down.
- *ThreeJS*: The code was long and depended on concepts that were likely less familiar (torus, specularity, Phong material, etc.). In our example `TorusGeometry` call, we intentionally left out the smoothing parameters that would need to be called (though we included an example of the smoothing parameter on the `SphereGeometry` example). To make it look right, developers needed to both choose the right shape (Torus) and modify the right parameters. This two-step process let developers mistakenly go down multiple wrong paths.

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

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

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

5 LIMITATIONS

There are several perspectives to consider limitations from. To start with, our theory itself takes a limited perspective on what tasks it considers when it defines API knowledge (for example excluding actually running the code or working in collaboration with others), which will require complementary or alternative theories with different perspectives (such as on the process of learning an API [20]). We also only considered three of many criteria to judge our theory by, but there are others that are likely be valuable. It also will take more time and the engagement of more people and perspectives (or a negative signal of their lack of engagement) to evaluate whether this theory is useful or true.

As part of our evaluation of our theory, we tested five hypotheses based on our theory, but those five hypotheses aren't necessarily the best representatives of our theory, and our experimental design might not be the best way to test those hypotheses. In particular, some participants complained about unclear or irrelevant code annotations, which might mean there were problems with how we wrote and presented the annotations separate from the theory. We also don't know which annotations participants looked at and whether they found the annotations that were relevant to the tasks they were working on or not. Additionally, participants mentioned a lack of time, and difficulty of tasks (particular with d3.js where many got stuck on the first task), so our task design could confound our results. The lack of time also complicates the focus of the participants, on whether to focus on understanding the API deeply, or quickly hacking together a fix, which are each legitimate goals in different situations, but makes generalizing our results more difficult. Additionally, we ran many statistical tests, which increases the likelihood of us finding statistically significant results by chance. Finally, the population of our study was from a large public university whose population differs in many ways from the global population of programmers or potential programmers [44].

6 DISCUSSION

6.1 Evaluations of our theory

Among the many ways a theory can be evaluated, we chose three to focus on: unifying prior findings into a coherent model, capturing internal logic and how parts relate, and producing (and testing) falsifiable claims. We have demonstrated how our theory unifies a number of previous findings (see 3.3.1) into a coherent model consisting of domain concepts, execution facts, and API usage patterns. We believe this model has clear internal logic and clear relationships between its pieces (see 3.3.2). Finally, we have made a start at testing hypotheses generated by our theory through a single user study.

In our study, we controlled access to the three components of knowledge from our theory, and we found mixed evidence for: 1) in specific situations, each component of knowledge significantly increased task progress and perceived API understanding, 2) that when these components of knowledge are lacking, people want it, and 3) that *access* to this knowledge is not always sufficient for progress and can be overwhelming. The mixed evidence in support of our theory, it does add nuances that demand further refinement. For example, it appears that learning concepts, facts, and patterns is not always straightforward and can be overwhelming. Finding the right piece of information at the right time is crucial, and while our study provided some mechanisms for searching, finding the right annotations still seemed difficult (at least within our limited time which gave little opportunity to learn deeply). This is consistent with prior work showing that API documentation can quickly become overwhelming and is not always easy to find when one needs to learn it [49]. This is no different from any other formal or informal learning context, where instruction needs to be tailored to a learner's prior knowledge.

6.2 Future work

Of course, there is substantially more work to do to with this theory, both in evaluating it (more empirical studies, as well as evaluations on other criteria such as how well it aids in answering questions in the field and what questions does it generate), as well as in refining its claims. For example: How well does current API documentation fit into our categories (we have already run a second study doing this with StackOverflow [48])? How well does our theory help developers and API designers categorize API knowledge? Are there other categories of knowledge that our theory misses? Our theory claims that a lack of *robust* knowledge results in task difficulties and defects, and we found mixed evidence for it. Is this true, and if so, by what mechanisms does robust API knowledge prevent defects? The theory also isn't clear on the effects of having robust knowledge of *part* of an API, but brittle or no knowledge of other parts. Are there cumulative benefits to having a broader knowledge base of robust knowledge about an API, or is the utility of knowledge highly task-specific? Finally, are there particular concepts, facts, and patterns about APIs that are more important than others to robust use of the API, or is it entirely task dependent? If so, why, and what would this mean for API documentation and tools? These unanswered questions are central to building a more powerful, predictive, and explanatory theory of API learning.

6.3 Implications

If we believe the theory, it has several implications for the design of API learning materials such as documentation, tutorials, Q&A, and classroom instruction. For example, the theory suggests that documentation should at the very least contain as many concepts, facts, and patterns known to be relevant as possible. We suspect most documentation does not, hence the popularity of sites like Stack Overflow. Second, our theory suggests that what *subset* of concepts, facts, and patterns a developer needs to know is highly-task dependent, suggesting that media like API documentation are rarely going to be structured in a way that optimally supports learning, both because they only offer one structure, and because their content is written to one level of prior knowledge. Media like tutorials, question answering sites, or even entirely new tools that attempt to retrieve and present relevant knowledge for a task and for a specific learner's prior knowledge are likely to be much more effective. Recent proposals for on-demand documentation [39], tools that have begun to automatically extract code patterns and their alternatives (e.g., [15]), and program analyses that can automatically extract execution facts about API behavior (e.g., [18], [53]), point the way to a future in which people learning an API can get exactly the knowledge they need for any given task. Our theory can help generate additional ideas for future tools, while also explaining how current tools do and do not support learning.

Until that future comes, our theory has also several implications for practice. Every day, millions of students, end-user programmers, and professional developers are encountering new APIs, trying to learn and use them productively. In the absence of great learning materials, learners try different strategies that can be related to our theory (e.g., opportunistic learners might be taking a API usage pattern first strategy, while systematic programmers might be taking a concept and fact first strategy [6, 7, 28]). By recognizing how their learning relates to our theory, learners could consider increase their awareness of the strengths and weaknesses of their strategy and make appropriate adjustments to gain all the knowledge they will end up needing or consider alternative strategies entirely.

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

ACKNOWLEDGMENTS

We'd like to thank Amanda Swearngin, Benji Xie, Alannah Oleson, Yim Register, Melissa Hovik, Lauren Bricker, Harrison Kwik, and Tal August for their feedback and help. Portions of this material are based upon work supported by the National Science Foundation under Grant No. 1703304, 1735123, and 1836813.

REFERENCES

[1] 2018. npm. https://www.npmjs.com/
[2] Miltiadis Allamanis and Charles Sutton. 2013. Why, when, and what: Analyzing Stack Overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 53–56. https://doi.org/10.1109/MSR.2013.6624004 ISSN: 2160-1852.
[3] Andrew Begel and Beth Simon. 2008. Struggles of new college graduates in their first software development job. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 226–230.

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

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

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

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

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

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

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

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

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

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

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

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

[17] Jane Hsieh, Michael Xieyang Liu, Brad A. Myers, and Aniket Kittur. 2018. An Exploratory Study of Web Foraging to Understand and Support Programming Decisions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 305–306.

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

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

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

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

[22] A. J. Ko, R. DeLine, and G. Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*. 344–353. https://doi.org/10.1109/ICSE.2007.45

[23] Amy J. Ko and Brad A. Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1âĂŞ2 (Feb. 2005), 41–84. https://doi.org/10.1016/j.jvlc.2004.08.003

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

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

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

[27] Wayne G. Lutters and Carolyn B. Seaman. 2007. Revealing actual documentation usage in software maintenance through war stories. *Information and Software Technology* 49, 6 (June 2007), 576–587. https://doi.org/10.1016/j.infsof.2007.02.013

[28] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2017. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication* (July 2017), 0047281617721853. https://doi.org/10.1177/0047281617721853

[29] Leo A Meyerovich and Ariel Rabkin. [n. d.]. Empirical Analysis of Programming Language Adoption. ([n. d.]), 18.

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

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

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

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

[34] Chris Parnin and Christoph Treude. 2011. Measuring API documentation on the web. ACM Press, 25–30. https://doi.org/10.1145/1984701.1984706

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

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

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

[38] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (Dec. 2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

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

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

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

[42] J. Sillito and A. Begel. 2013. App-directed learning: An exploratory study. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 81–84. https://doi.org/10.1109/CHASE.2013.6614736

[43] E. Soloway and K. Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (Sept. 1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[44] Christian Sturm, Alice Oh, Sebastian Linxen, Jose Abdelnour Nocera, Susan Dray, and Katharina Reinecke. 2015. How WEIRD is HCI?: Extending HCI Principles to other Countries and Cultures. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2425–2428. http://dl.acm.org/citation.cfm?id=2702656 00000.

[45] J. Stylos and B. A. Myers. 2006. Mica: A Web-Search Tool for Finding API Components and Examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 195–202. https://doi.org/10.1109/VLHCC.2006.32

[46] Patrick Suppes. 1967. What is a Scientific Theory? *Philosophy of Science Today* (1967), 55–67.

[47] Kyle Thayer and Amy J. Ko. 2017. Barriers Faced by Coding Bootcamp Students. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 245–253. https://doi.org/10.1145/3105726.3106176

[48] Kyle Matthew Thayer. 2020. *Practical Knowledge Barriers in Professional Programming*. Thesis. https://digital.lib.washington.edu:443/researchworks/handle/1773/45471 Accepted: 2020-04-30T17:42:10Z.

[49] G. Uddin and M. P. Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (July 2015), 68–75. https://doi.org/10.1109/MS.2014.80

[50] A. Von Mayrhauser and A.M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (Aug. 1995), 44–55. https://doi.org/10.1109/2.402076

[51] Robert Bennett Watson. 2015. *The effect of visual design and information content on readersâĂŹ assessments of API reference topics.* PhD Thesis.

[52] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[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 (ICSE '12).* IEEE Press, Piscataway, NJ, USA, 826–836. http://dl.acm.org/citation.cfm?id=2337223.2337321

A SUPPLEMENTAL MATERIALS

A.1 Study Task Descriptions

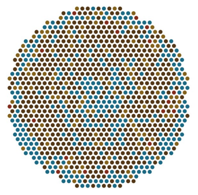
On the next two pages are a summary of the tasks we gave to participants in our study, along with the subtasks we used to evaluate the participants' progress.

d3.js

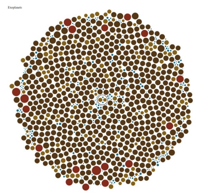
Exoplanet Visualization

Start/example code sources: https://bl.ocks.org/mbostock/3007180 https://beta.observablehq.com/@mbostock/d3-circle-packing

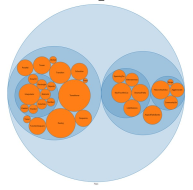
Start Output



Goal Output



Example Code Output



Task

This text was shown to participants.

Subtasks

These were not shown to participants. They were used by researchers to measure progress.

1) Add title to the top-left that says "Exoplanets" (see Final Graph below)
1. Use the svg variable
2. Call append()
3. Call text()
4. Call attr() for dx,dy
5. Position correctly

2) Make the circles have an area proportional with the square of the planet radius. Make sure the circles stay close together (reduce big spaces).
1. Change attr("r")
2. Use d.r or d.data.radius in some way
3. To d.r or d.data.radius (allow for constant multipliers)
4. Modify sum()
5. To radius * radius

3) Add the planets of our solar system to the graph (the Array.concat function may help). It is difficult to tell if this worked, so if you think you have it, go to the next task.
1. Concat planets to array

4) Put planets of our solar system in a circle, separated from the rest of the planets. Make the circle surrounding the planets of our solar system not be filled in (you can use the provided "hollow-circle" class in the css).
1. Make separate child in hierarchy for our solar system
2. Use class for circle

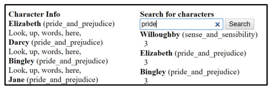
5) Sort planets by distance to the sun. That is, planets in the middle are close to the sun, planets on the outside are far from the sun.
1. Use sort function on D3 hierarchy
2. Correctly perform sort (including handling children and NANs)

Natural

Book Character Word Association

Example code source: https://github.com/NaturalNode/natural

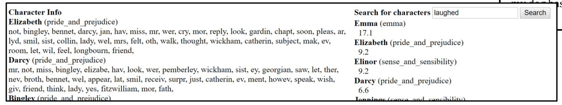
Start Output



Character Info
Elizabeth (pride_and_prejudice)
Look, up, words, here,
Darcy (pride_and_prejudice)
Look, up, words, here,
Bingley (pride_and_prejudice)
Look, up, words, here,
Jane (pride_and_prejudice)

Search for characters pride Search
Willoughby (sense_and_sensibility)
3
Elizabeth (pride_and_prejudice)
3
Bingley (pride_and_prejudice)
3

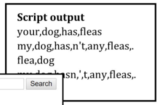
Goal Output



Character Info
Elizabeth (pride_and_prejudice)
not, bingley, bennet, darcy, jan, hav, miss, mr, wer, cry, mor, reply, look, gardin, chapt, soon, pleas, ar, lyd, smil, sist, collin, lady, wel, mrs, felt, oth, walk, thought, wickham, catherin, subject, mak, ev, room, let, wil, feel, longbourn, friend,
Darcy (pride_and_prejudice)
mr, not, miss, bingley, elizabe, hav, look, wer, pemberley, wickham, sist, ey, georgian, saw, let, ther, nev, broth, bennet, wel, appear, lat, smil, receiv, surpr, just, catherin, ev, ment, howev, speak, wish, giv, friend, think, lady, yes, fitzwilliam, mor, fath,
Bingley (pride_and_prejudice)

Search for characters laughed Search
Emma (emma)
17.1
Elizabeth (pride_and_prejudice)
9.2
Elinor (sense_and_sensibility)
9.2
Darcy (pride_and_prejudice)
6.6
Jennings (sense_and_sensibility)

Example Code Output



Script output
your,dog,has,fleas
my,dog,has,n't,any,fleas,.
flea,dog
sn,',t,any,fleas,.

1) getWordsNextToCharacterName currently returns words that occur up to two before and two after a character name, excluding the name itself (i.e., "word1 word2 characterName word1 word2"). You want to modify it to find words that occur up to six words before or six words after a character name. Check the results in the debugging console.
*1. Modify the trigrams() line
2. Use the ngrams() function
3. Use ngrams(, 7)

2) You want to find out which words commonly appear near a character's name (relative to words near other characters' names). For example: "she" often appears near the name Elizabeth, and more so than near other character names. Replace the code in findWordsForCharacters that sets ["Look", "up", "words", "here"] with code that finds those commonly appearing words (you may have to modify other code as well). To do this, use the Natural library to group the strings returned by getWordsNextToCharacterName and find those words.
*1. Use tfidf in some way
2. Add documents
3. Add ngrams from each character as a document
4. Call listterms()
5. Of character i
6. Try to print it
7. Using charwords[j].term

3) You want to be able to search for characters that best fit words. Use your work from task 2 and the Natural library to find how commonly the search words appear near a character's name (relative to other characters). Fill in the match value in the search function. Then search should work like this:
1. Make tfidf global (or redo work)
2. Replace the match value
3. With result of tfidf.tfidf() function
4. Passing it (searchTerms, i)

4) You notice some words found aren't ones you are interested in (like "she" and "not"). Use the Natural library to determine the type of word found based on the context of the group of seven words you found it with. Then ignore the following types of words:
- **Coordinating Conjunctions, like "nor"**
- **Determiners, like "the"**
- **Preposition or subordinating conjunctions, like "towards"**
- **Personal pronouns, like "himself"**
- **Possessive pronouns, like "our"**

1. Set up rules / lex / tagger
2. Call tagger.tag
3. Of ngram words
4. Do for loop over tags
5. Get the part of speech value
6. Check if it is not some parts of speech
7. Specifically PRP PRP$ IN DT CC

5) You notice that searching for "laugh" and "laughed" brings up different results, but you think those should be combined. Use the Natural library to combine related words like "laugh" and "laughed". Make sure search works with this.
1. Call the stem() function
2. Stem the ngram words before tfidf add documents
3. Stem words user entered before searching

Fig. 8. Task sets for d3.js and Natural APIs. Shows the output of the starter code participants were given, the final output they were asked to produce, and the output of the example code they were given as a reference. Task text participants were given are in bold (intermediate goal images were shown with some tasks, but are not shown here). The subtasks that were used by researchers to measure progress are in light typeface between the tasks. Substasks used as bottlenecks (section 4.2.2) are highlighted with a star.

OpenLayers

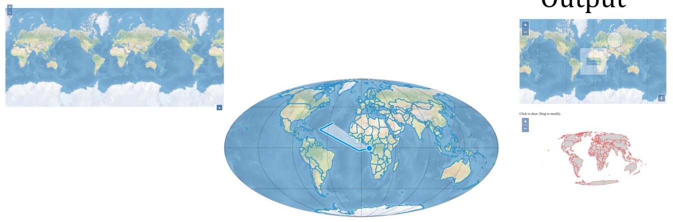
Interactive Map

Start/example code source: https://openlayers.org/en/latest/examples/

Start Output

Goal Output

Example Code Output



1) Change your map to be a rounded map using the sphereMollweideProjection variable already defined. If it works, you should see ridges along the top of the map.
1. Set projection to sphereMollweideProjection
2. Set view projection to sphereMollweideProjection

2) Add lines representing latitude and longitude to the map.
*1. Use graticule
2. Correctly store map to variable and use in graticule call

3) Add outlines to the countries from: https://openlayers.org/en/v4.6.4/examples/data/geojson/countries-110m.geojson (url in example code). Make sure you don't cover the countries with any color, but just have the outlines. Those outlines should be color #3399CC and have a width of 1.5.
1. Create source vector
2. With url and GeoJson
3. Create vector layer
4. With source vector (make sure vector is defined first so this isn't undefined)
5. Set style
6. Fill correctly (or have no fill style)
7. Stroke correctly (color and width)
8. Add vector layer to map

4) Cover the areas outside the main globe (marked with latitude/longitude lines) with white so that you just see the main oval of the earth. Don't worry about covering the outside area perfectly, just get it close.
1. Create linear rings
2. Circle
3. Resized to ellipse
4. Rectangle
5. Create polygon
6. With Rectangle then circle
7. Sized correctly
8. Create vector source
9. With polygon
10. Create vector layer
11. With vector source
12. Style it white
13. Add layer to map

5) Allow users to draw shapes and modify them. Don't let them modify the country borders or any other feature. (Use OpenLayers default colors for the drawn shapes).
1. New layer
2. Interaction - modify
3. Interaction - draw

6) To aide with drawing, make it so the map makes it easier to draw points along the edges and corners of countries and previously drawn shapes.
1. Add snap to new layer
2. Add snap to country layer

Task

This text was shown to participants.

Subtasks

These were not shown to participants. They were used by researchers to measure progress.

ThreeJS

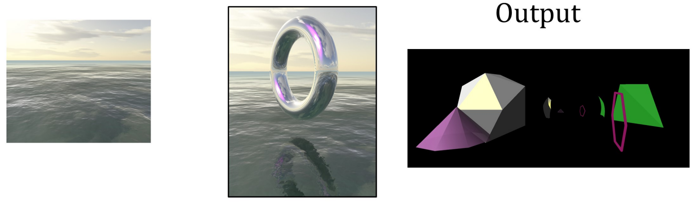
Spinning Loading Ring

Start/example code sources: https://threejs.org/examples/ https://threejs.org/examples/#webgl_shaders_ocean

Start Output

Goal Output

Example Code Output



1) Add this shape. Make it sized with an overall diameter of 40, the tube part with a diameter of about 10, and position the bottom 10 above the water. It should look smooth and circular. See the next page for clearer views of the shape.
1. Create any geometry
2. Create material (give point if they make a mesh with no material)
3. Create mesh from the geometry and material
4. Add to scene
*5. TorusGeometry()
6. Sized correctly (20,5,...)
*7. Smooth (fourth parameter)
8. Position changed
9. Positioned correctly (position.y = 30)

2) Make the shape be gray (hex color: 444444) and have a purple shine (hex color: 991199).
1. Phong Material
2. setting emissive instead of color
3. Color correct (in emissive)
4. Specular set (Shininess set optionally)
5. To correct color

3) Make the shape appear perfectly reflective (note: the reflections can be hard to see if your shape is too dark, so make it white [hex color: ffffff] and uniformly shaded). Make sure this reflection work for both the sky and ocean.
1. Color white
2. That white color should be set in emmisive
3. Reflect camera
4. Phong material
5. With env map
6. Set cube reflection mapping
7. Set camera position to match torus
8. Update camera in animate

4) Make the doughnut spin around it's vertical axis to indicate that the website is loading.
1. Get current time
2. Change object rotation based on current time
3. Specifically change rotation.y

Fig. 9. Task sets for OpenLayers and ThreeJs APIs. Shows the output of the starter code participants were given, the final output they were asked to produce, and the output of the example code they were given as a reference. Task text participants were given are in bold (intermediate goal images were shown with some tasks, but are not shown here). The subtasks that were used by researchers to measure progress are in light typeface between the tasks. Substasks used as bottlenecks (section 4.2.2) are highlighted with a star.

A.2 Participant Comment Qualitative Coding Codebook

To code the participant comments, we created codes based on the category (and sometimes sub-category) of what a participant was referring to, as well as a set of modifiers for how they were talking about the category.

A.2.1 Modifier Codes.

- **-U (Useful or Used):** A participant mentioned something being useful or mentioned using something. Generally, responses to our question, "*What information or strategies did you find most helpful?*" fall into this category. E.g., "*The most helpful thing here was just messing with the code*" (P46, Natural).
- **-W (Wanted):** A participant mentioned something they wanted to have but didnâĂŹt. Generally, responses to our questions, "*What additional information would you have wanted?*" fall into this category. E.g., "*I think I would've enjoyed more of a step by step on what exactly each line was doing*" (P46, OpenLayers).
- **-X (Useless or Unused):** A participant mentioned something as not being helpful or not using something. E.g., "*I didn't look at the annotations very much*" (P60, OpenLayers), "*Code was difficult to understand.*" (P19, D3)
- **-Sp (Specific):** A participant mentioned a specific example that falls within one of the categories but doesn't mention the category itself. We did not use this in our analysis.
- ∼ **(Equivocating):** If a participant equivocated or moderated in their response, we added a "∼". E.g., "*The annotations and examples were only relatively helpful*" (P47, D3).

A.2.2 Category codes.

- **Annotations:**
  - **C (Concepts):** A mention of domain concepts or concept annotations. E.g., C-U: "*The FACT and CONCEPT cards were very helpful in understanding some things such as a 'pack.'*" (P29, D3). C-W-sp: "*[I wanted] a more thorough explanation of tfidf (how it works, different applications, etc.)*" (P14, Natural).
  - **T (Templates):** A mention of templates or API usage patterns. E.g., T-U: "*The template for creating an object and add it to a scene is helpful*" (P67, ThreeJS). T-W-sp: "*Rather than just giving what one specific function means, but rather what the group of code does holistically (chunks) like how to append text*" (P48, D3).
  - **F (Facts):** A mention of execution facts or our fact annotations. E.g., F-X: "*Facts were not very useful w/o the templates*" (P4, D3). F-W-sp: "*Maybe [I'd want] a brief listing of the tools and functions from the library*" (P1, Natural).
  - **A (Annotations):** A vague mention of annotations that we can't deduce based on what the participant had access to for that task. E.g., A-W: "*I would have liked some annotations on the example code like we had before*" (P28, Natural).
- **Documentation:**
  - **TD (Task Description):** A mention of our instructions for the task. E.g., TD-U: "*[I found the] clear objectives and starting point [helpful]*" (P21, Natural).
  - **OD (Overview Document):** A mention of wanting an overview document, or high-level information. E.g., OD-W: "*I would have wanted a high-level overview of the capabilities and features of the library. What is a vector? What is a layer? etc.*" (P61, OpenLayers).
  - **Tu (Tutorial, steps to do something):** A mention of a tutorial on the library or code. E.g., Tu-W: "*[I would want] A tutorial on modifying maps using open layers*" (P28, OpenLayers).
  - **HT (How to):** A mention of a general "how to.." without us being able to categorize it more specifically. E.g., Ht-W: "*[I would want] How to clear off the polygons on the map once I drew them*" (P43, OpenLayers).
  - **D (documentation):** A vague mention of "documentation" or generic information by a participant. E.g., D-W: "*Having more explanation into how to use the API would have been helpful*" (P36, OpenLayers).
- **Programming Code:**
  - **EC (Example Code):** A mention of the example code we provided for the tasks. E.g., EC-U: "*the sample code was useful especially because of having different variety of images & shapes*" (P20, ThreeJS).

  * **TC (Task Code):** A mention of the code participants were to modify for their task (sometimes referred to as "starter code" or possibly "source code"). E.g., TC-U: "*The code in the task was also helpful*" (P61, ThreeJS).
  * **Co (Code):** A mention of code without it being clear which code they are referring to. E.g., Co-X: "*Code was difficult to understand*" (P19, D3).
  * **CP (Copy/Paste):** A mention of copying and pasting code. E.g., CP-U: "*I mainly just copied / pasted code and tweaked what was needed*" (P17, ThreeJS).
  * **-VN- (Variable Names):** A mention of variable names within the code. This acts as a subcategory of any of the above code types. E.g., TC-VN-U: "*Some names in the source code were helpful to make educated guesses about how I might complete a task (i.e. 'r' probably having something to do with a radius)*" (P21, D3).
  * **-FN- (Function Names):** A mention of function names within the code. This acts as a subcategory of any of the above code types. E.g., Co-FN-U: (which we shortened to just FN-U): "*In general, the function names in THREE was very intuitive*" (P16, ThreeJS).
  * **-Cm- (Comments):** A mention of comments within the code. This acts as a subcategory of any of the above code types. E.g., EC-Cm-W and TC-cm-W: "*Comments would be useful (both sets of codes)*" (P4, D3).
- **Other:**
  * **N (Nothing):** A mention of nothing. E.g., N-U "*nothing was remotely useful*" (P48, D3).
  * **Ti (Time):** A mention of time. E.g., Ti-W: "*I just needed more time*" (P61, Natural).
  * **TMI (Too Much Information):** A mention of their being too much information. E.g., "*Because there were so much information and no clues to figure out how the info can be used for a specific task?*" (P16, Natural).
  * **S (Search):** A mention of using search features (like ctrl-f for the browser search). E.g., S-U: "*I used ctrl+F to find the function name and tried to modify the numbers related to change something for task*" (P16, Natural).
  * **PK (Previous knowledge):** A mention of prior knowledge (whether present or missing). E.g., PK-U: "*I had some knowledge about Ngram, so I know what they do*" (P67 Natural).
  * **PL (Programming Language):** A mention of knowledge of the underlying programming language. E.g., PK-W: "*I wish I could've googled things since I haven't used JS or HTML in a while & couldn't remember how to write the tags*" (P38, D3).
  * **Exp (Experimentation):** A mention of experimenting with changing the code. E.g., Exp-U: "*Using the example and experimenting were helpful*" (P46, D3).
  * **Db (Debugger):** A mention of using a debugger. E.g., Db-U: "*[I found it helpful to use] Google chrome's helper info regarding the functions of a given thing you type*" (P60, D3).

A.3 Study Participant Instructions

On the next six pages are the instructions that we gave to participants as they went through their tasks, including the instructions for the OpenLayers API, but removing the equivalent instructions for ThreeJS, d3.js, and Natural (the full set of instructions are in the supplemental materials and at https://github.com/kylethayer/API-Knowledge-Theory-Supplemental-Materials).

Experiment Condition _____

Welcome to the JavaScript Library Experiment!

Programming Experience

(These answers will be stored with your task progress and used in our analysis)

Approximately how many programming languages have you learned (that is, written at least one program in)? (Circle one)

1 2-4 5-9 10+

Approximately how many programming libraries have you learned (that is, have used at least once)? (Circle one)

1 2-4 5-9 10+

How confident do you feel debugging JavaScript in Chrome? (Circle one)

Very Uncomfortable *Very Comfortable*

1 2 3 4 5 6 7

Demographic Information

(These answers will only be used for reporting summary characteristics of all participants and will be stored separately from all other data)

Age:

Gender:

Year or stage in school:

Email Updates

(Email addresses will be stored separately and will only be used emailing updates and will be stored separately from all other data)

If you would like to receive updates on the research, including full solutions to the tasks once the research is over, put your email below:

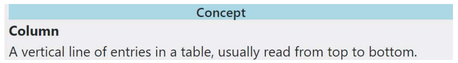
(Optional) Email:

Instructions

You will work on tasks in four different libraries. You will have 15 minutes to work on each API. Your goal while working on the tasks is not just to solve the them, but also understand how that solution works. We've designed the tasks to be very difficult without the expectation of anyone finishing them all or even getting close, so just complete as many as you can in order. When you complete a task, circle it with a pen. We'd also like a snapshot of your solution to each task, so when you complete a task, right click on the js file and select duplicate to make a copy and continue editing the original (see further details on the next page).

We have developed learning resources for these libraries in the form of code annotations. Depending on the task you will see the following three types of annotations:

- Concept Definitions (**Concepts**): General concepts used by a code library.

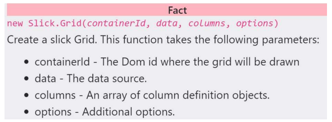


Concept

Column

A vertical line of entries in a table, usually read from top to bottom.

- Execution and Usage Facts (**Facts**): Facts about how the library will run and how to make calls to the library.



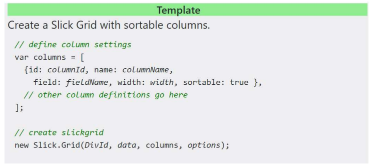
Fact

```
new Slick.Grid(containerId, data, columns, options)
```

Create a slick Grid. This function takes the following parameters:

- containerId - The Dom id where the grid will be drawn
- data - The data source.
- columns - An array of column definition objects.
- options - Additional options.

- Code Templates (**Templates**): Multiple lines of code that are used together to achieve some output with the library.



Template

Create a Slick Grid with sortable columns.

```
// define column settings
var columns = [
  {id: columnId, name: columnName,
    field: fieldName, width: width, sortable: true },
  // other column definitions go here
];

// create slickgrid
new Slick.Grid(DivId, data, columns, options);
```

We want to see how useful these resources are, so you can't use any other resources like Google or Stack-Overflow. Also, please try not to read from other participants' screens.

For each task, you will be given access to:

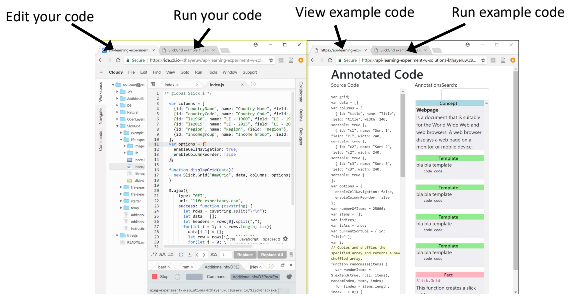
- The starting source code, which you will modify
- Example code, possible with some or many annotations. You may copy from this code.
- The running version of the example code

After working on each API, we will ask you how well you think you know each line of code in the example code, and ask what each line of code does. Then, when it is time to start the next task, you will minimize the left and right browser windows and begin.

After you have finished the study, do not talk about the tasks or Libraries with potential future participants until we have completed the study for everyone.

Practice API

You will see two browser windows open on your screen.



Left Browser window:

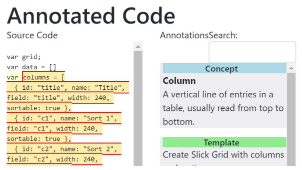
- The first tab has a JavaScript file open in cloud9. This is where you will edit files. Make sure to save them (ctrl-s) when you want to see your changes.
- The second tab has the running version of the code you are editing. On this tab use ctrl + refresh button to reload the page after you have finished editing.

Right Browser window:

- The first tab has annotated example code. You can click on highlighted code to view annotations for that code.
- The second tab has the running example code. Look at this to gain insights on what the example code does. Depending on the task you may have few or no annotations.

Practice Tasks

1) Click on the annotation for setting the columns variable. Scroll up and down the annotation column to see all the annotations. Click elsewhere to un-highlight the annotations.



Annotated Code

Source Code

```
var grid;
var data = []
var columns = [
  { id: "title", name: "Title",
field: "title", width: 240,
sortable: true },
  { id: "c1", name: "Sort 1",
field: "c1", width: 240,
sortable: true },
  { id: "c2", name: "Sort 2",
field: "c2", width: 240,
```

AnnotationsSearch:

Concept

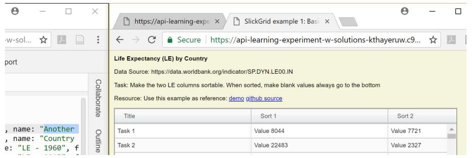
Column

A vertical line of entries in a table, usually read from top to bottom.

Template

Create Slick Grid with columns

2) Open the second tab on the right browser to see the working version of the annotation code.



```
, name: "Another
, name: "Country
e: "LE - 1960", f
```

https://api-learning-experiment-w-solutions-kthayeruw.c9...

Life Expectancy (LE) by Country

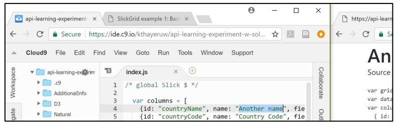
Data Source: https://data.worldbank.org/indicator/SP.DYN.LE00.IN

Task: Make the two LE columns sortable. When sorted, make blank values always go to the bottom

Resource: Use this example as reference: demo github source

Title	Sort 1	Sort 2
Task 1	Value 8044	Value 7721
Task 2	Value 22483	Value 2327

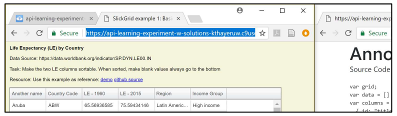
3) In the task.io code, on line 4 change `name:  "Country Name"` to be `name: "Another Name"`. Save the change (ctrl-s) and refresh the second tab to see the change take effect.



https://ide.c9.io/kthayeruw/api-learning-experiment-w-sol...

```
/* global Slick $ */

var columns = [
  {id: "countryName", name: "Another name", fie
  {id: "countryCode", name: "Country Code", fie
```



https://api-learning-experiment-w-solutions-kthayeruw.c9us

Life Expectancy (LE) by Country

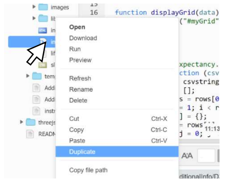
Data Source: https://data.worldbank.org/indicator/SP.DYN.LE00.IN

Task: Make the two LE columns sortable. When sorted, make blank values always go to the bottom

Resource: Use this example as reference: demo github source

Another name	Country Code	LE - 1960	LE - 2015	Region	Income Group
Aruba	ABW	65.56936585	75.59434146	Latin Americ...	High income

4) Right click on the index.js file and select duplicate to make a snapshot copy of the file.

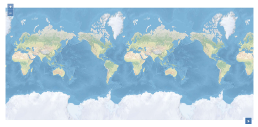


5) Tell the researcher that you are ready.

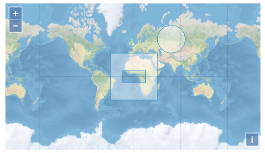
6) You can try editing the file more or reading the annotations if you want. Don't open any other files. Otherwise, just wait until everyone is ready.

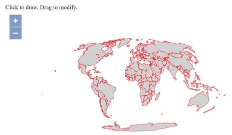
Open Layers

You are creating an interactive map on a webpage. To test if this mapping library will work for you project, you want to create a nice-looking interactive map that people can draw shapes on. You start with a map of the world:



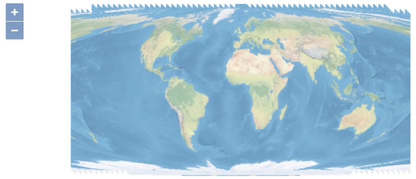
You have found example code that creates the following two maps:



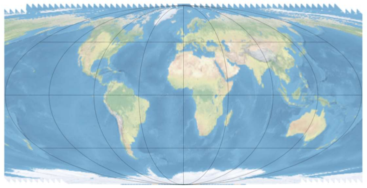


Tasks: (Do these in order. When you've finished a task, circle the number and duplicate the js file. Continue working on the original js file.)

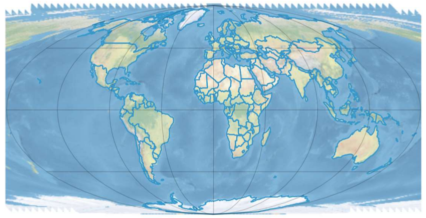
1) Change your map to be a rounded map using the sphereMollweideProjection variable already defined. If it works, you should see ridges along the top of the map.



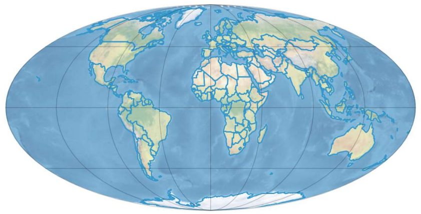
2) Add lines representing latitude and longitude to the map.



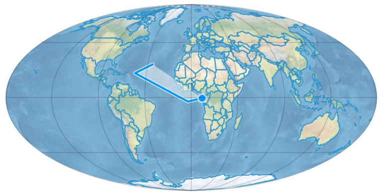
3) Add outlines to the countries from: https://openlayers.org/en/v4.6.4/examples/data/geojson/countries-110m.geojson (url in example code). Make sure you don't cover the countries with any color, but just have the outlines. Those outlines should be color #3399CC and have a width of 1.5.



4) Cover the areas outside the main globe (marked with latitude/longitude lines) with white so that you just see the main oval of the earth. Don't worry about covering the outside area perfectly, just get it close.



5) Allow users to draw shapes and modify them. Don't let them modify the country borders or any other feature. (Use OpenLayers default colors for the drawn shapes).



6) To aide with drawing, make it so the map makes it easier to draw points along the edges and corners of countries and previously drawn shapes.

Instructions:

Quickly, without looking at your computer screen, circle the digits in the understanding columns to rate the following statement for each substantive line of code. Answer the two questions for as many lines as you can, starting at the top.

The questions are:

1) **I understand what this line does and its purpose in the larger program.**
   (**1** for completely disagree, **5** for completely agree)

   (**1** for completely disagree, **5** for completely agree)

	Understand	Source Code of the Example From Your Task
1	1 2 3 4 5	`var map1 = new ol.Map({`
2	1 2 3 4 5	`  target: 'map1',`
3	1 2 3 4 5	`  view: new ol.View({`
4	1 2 3 4 5	`    center: [0, 0],`
5	1 2 3 4 5	`    zoom: 1,`
6		`  }),`
7	1 2 3 4 5	`  layers: [`
8	1 2 3 4 5	`    new ol.layer.Tile({`
9	1 2 3 4 5	`      source: new ol.source.TileWMS({`
10	1 2 3 4 5	`        projection: 'EPSG:4326',`
11	1 2 3 4 5	`        url: 'http://demo.boundlessgeo.com/geoserver/wms',`
12	1 2 3 4 5	`        params: {`
13	1 2 3 4 5	`          'LAYERS': 'ne:NE1_HR_LC_SR_W_DR'`
14		`        }`
15		`      })`
16		`    })`
17		`  ]`
18		`});`
19		
20	1 2 3 4 5	`new ol.Graticule({`
21	1 2 3 4 5	`  map: map1`
22		`});`
23		
24	1 2 3 4 5	`var circle = new ol.geom.Circle([8e6,8e6], 3e6);`
25	1 2 3 4 5	`var circlePoly = ol.geom.Polygon.fromCircle(circle, 15);`
26		
27	1 2 3 4 5	`var squarePoly = new ol.geom.Polygon();`
28	1 2 3 4 5	

		`squarePoly.appendLinearRing(new ol.geom.LinearRing([[5e6, 5e6], [5e6, -5e6], [-5e6, -5e6],`
29	1 2 3 4 5	`[-5e6, 5e6]]]));`
30	1 2 3 4 5	`var squareLinearRing = squarePoly.getLinearRing(0);`
31	1 2 3 4 5	`squareLinearRing.scale(.5,.25);`
32		`squarePoly.appendLinearRing(squareLinearRing);`
33	1 2 3 4 5	
34	1 2 3 4 5	`var vectorSource = new ol.source.Vector({`
35	1 2 3 4 5	`  features: [`
36	1 2 3 4 5	`    new ol.Feature(circlePoly),`
37		`    new ol.Feature(squarePoly),`
38		`  ]`
39		`});`
40	1 2 3 4 5	
41	1 2 3 4 5	`var vectorLayer = new ol.layer.Vector({`
42		`  source: vectorSource`
43		`});`
44	1 2 3 4 5	
45		`map1.addLayer(vectorLayer);`
46		
47		
48	1 2 3 4 5	
49	1 2 3 4 5	`var countrySource = new ol.source.Vector({`
50	1 2 3 4 5	`  url: 'https://openlayers.org/en/v4.6.4/examples/data/geojson/countries-110m.geojson',`
51		`  format: new ol.format.GeoJSON()`
52		`});`
53	1 2 3 4 5	
54	1 2 3 4 5	`var countryLayer = new ol.layer.Vector({`
55	1 2 3 4 5	`  source: countrySource,`
56	1 2 3 4 5	`  style: function(feature) {`
57	1 2 3 4 5	`      return new ol.style.Style({`
58	1 2 3 4 5	`        fill: new ol.style.Fill({`
59		`          color: 'lightgray'`
60	1 2 3 4 5	`        }),`
61	1 2 3 4 5	`        stroke: new ol.style.Stroke({`
62		`          color: 'rgba(256,0,0,.7)'`
63		`        })`
64		`      });`
65		`    }`
66		`});`
67	1 2 3 4 5	
68	1 2 3 4 5	`proj4.defs('ESRI:53009', '+proj=moll +lon_0=0 +x_0=0 +y_0=0 +a=6371000 ' +`
69	1 2 3 4 5	`  '+b=6371000 +units=m +no_defs');`
70	1 2 3 4 5	`var sphereMollweideProjection = new ol.proj.Projection({`
71	1 2 3 4 5	`  code: 'ESRI:53009',`
72	1 2 3 4 5	`  extent: [-18000000, -9100000,`

73	1 2 3 4 5	`    18000000, 9200000],`
74		`  worldExtent: [-179, -89.99, 179, 89.99]`
75		`});`
76	1 2 3 4 5	
77	1 2 3 4 5	`var map2 = new ol.Map({`
78	1 2 3 4 5	`  keyboardEventTarget: document,`
79	1 2 3 4 5	`  layers: [countryLayer],`
80	1 2 3 4 5	`  target: 'map2',`
81	1 2 3 4 5	`  view: new ol.View({`
82	1 2 3 4 5	`    center: [0, 0],`
83	1 2 3 4 5	`    projection: sphereMollweideProjection,`
84		`    zoom: 1`
85		`  })`
86		`});`
87	1 2 3 4 5	
88	1 2 3 4 5	`map2.addInteraction(new ol.interaction.Modify({`
89		`  source: countrySource`
90		`}));`
91	1 2 3 4 5	
92	1 2 3 4 5	`map2.addInteraction(new ol.interaction.Draw({`
93	1 2 3 4 5	`  type: 'Polygon',`
94		`  source: countrySource`
95		`}));`
96	1 2 3 4 5	
97	1 2 3 4 5	`map2.addInteraction(new ol.interaction.Snap({`
98		`  source: countrySource`
		`}));`

Instructions:

Write one or more sentences in answer to the following two questions. You may reference your code and the example code from this task.

1) What information or strategies did you find most helpful?

2) What additional information would you have wanted?

accepted Dec 2020