Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming

Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich

Institute for Software Research
School of Computer Science
Carnegie Mellon University
`{sunshine, jdh, aldrich}@cs.cmu.edu`

Abstract. Application Programming Interfaces (APIs) often define object protocols. Objects with protocols have a finite number of states and in each state a different set of method calls is valid. Many researchers have developed protocol verification tools because protocols are notoriously difficult to follow correctly. However, recent research suggests that a major challenge for API protocol programmers is effectively searching the state space. Verification is an ineffective guide for this kind of search. In this paper we instead propose Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. We compare Plaiddoc to a Javadoc control in a between-subjects laboratory experiment. We find that Plaiddoc participants complete state search tasks in significantly less time and with significantly fewer errors than Javadoc participants.

1 Introduction

Many Application Programming Interfaces (APIs) define object protocols, which restrict the order of client calls to API methods. Objects with protocols have a finite number of states and in each state a different set of method calls is valid. Protocols also specify transitions between states that occur as part of some method calls. A client of such a library must be aware of the protocol in order to use it correctly. For example, a file may be in the open or closed state. In the open state, one may read or write to a file, or one may close it, which causes a state transition to the closed state. In the closed state, the only permitted operation is to (re-)open the file.

Files provide a simple example of states, but there are many more examples. Streams may be open or closed, iterators may have elements available or not, collections may be empty or not, and even lowly exceptions can have their cause set, or not. More than 8% of Java Standard Library classes and interfaces define protocols, which is more than three times as many as define type parameters [1].

Protocols are implemented in mainstream languages like Java with low-level constructs: the state of an object is tracked with boolean, integer, or enum fields;

violations are checked explicitly and cause runtime exceptions like IllegalStateException; and constraints are specified in prose documentation. It is perhaps unsurprising, therefore, that APIs with protocols are difficult to use. In a study of problems developers experienced when using a portion of the ASP.NET framework, three quarters of the issues identified involved temporal constraints [19]. Three recent security papers have identified serious vulnerabilities in widely used security applications resulting from API protocol violations [14, 4, 28].

Many researchers have developed protocol checkers which are designed to make it easier for programmers to correctly use APIs with protocols (e.g. [3, 10, 13]). These tools require programmers to specify protocols using alias and typestate annotations that are separate from code. To automate the annotation process, several tools mine protocol specifications using dynamic analysis [8] or static analysis [2, 36]. A recent survey of automated API property inference techniques described 35 inference techniques for ordering specifications [24].

However, the qualitative studies described in [31, ch.3] found that programmers using API protocols spend their time primarily on four types of searches of the protocol state space. Protocol checker output is unlikely to help programmers perform many of these searches.

Instead, in this paper we introduce a novel documentation generator called Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. Plaiddoc is extracted automatically from the standard Javadoc annotations plus new Plaiddoc specifications. Plaiddoc is named for the Plaid programming language [32], which embeds similar state-oriented features, and from which Plaiddoc could, in principle, be automatically generated. We evaluate Plaiddoc against a Javadoc control in a 20-participant between-subjects laboratory experiment.

The experiment attempts to answer the following five research questions:

RQ1 Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc?
RQ2 Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?
RQ3 Will programmers who use Plaiddoc answer state search questions more correctly than programmers who use Javadoc?
RQ4 Will programmers get better at answering state search questions as they get more practice?
RQ5 Are programmers who use Plaiddoc better than programmers who use Javadoc at mapping general state concepts to API details?

All of the tasks performed by participants asked participants to answer a question. We therefore use the words task and question interchangeably in the rest of this paper. Most of these questions were instances of four state search categories discovered in two earlier, qualitative studies [31]. Some of the questions were not state related and were chosen to benefit Javadoc. Task ordering was alternated to measure learning effects, and a post-study quiz was administered to gauge concept understanding.

Participants using Plaiddoc completed state tasks in 46% of the time it took Javadoc participants, but were approximately equally fast on non-state tasks. Plaiddoc participants were also 7.6x less likely to answer questions incorrectly than Javadoc participants. Finally, Plaiddoc and Javadoc participants were approximately equally able to map state concepts to API details. Nevertheless, our overall results suggest that Plaiddoc can provide a lightweight mechanism for improving programmer performance on state-related tasks without negatively impacting traditional tasks.

More broadly, the results of this study also provide indirect support for several programming language design choices. This study provides quantitative evidence for the productivity benefits of type annotations as documentation and state-oriented language features.

2 Background and Related Work

The seminal paper entitled "Why a diagram is (sometimes) worth ten thousand words," [21] introduces a computational model of human cognition to compare informationally equivalent diagrams and text. They demonstrate in this model that solving math and physics problems with text-based information can require many more steps than solving the same problems with diagrams. The most important difference between the diagram steps and text steps is that much more effort in text is spent searching for needed details. One particularly noteworthy reason for the search difference is that diagrams often collocate details that are needed together.

Larkin and Simon's theory has been effectively applied to many other (non-diagramatic) information contexts. For example, Chandler shows in a series of experiments that integrated instructional material and the removal of non-essential material can facilitate learning in a variety of educational settings [5] . There are many more closely related examples: Green [15] develops cognitive dimensions to evaluate visual programming languages, the GOMS [20] model has proven effective at predicting user response to graphical user interfaces (GUIs), and MCRpd [34] models physical representations of digital objects.

The results of two studies of API design choices are best understood through Larkin and Simon's search lens. It is easier for programmers to use constructors to create instances than factory methods, because constructors are the default and are therefore the start of any search [11]. Methods that are located in the class a programmer starts with are easier to find than methods in related classes [30]. The impact of small design changes shown in these papers emphasizes the importance of information seeking on API usability, and suggests that a similar impact may be possible with other small interventions.

All of this research suggests that there is an opportunity to modify an API artifact to create an informationally equivalent alternative that will improve programmer performance with protocol search. Which artifact? Which changes will be most effective? To answer these questions it is useful to look at the interventions that have proven effective with other complex APIs.

One effective way to learn to use an API is to find a related example. A study of programmers using reusable Smalltalk GUI components and found that participants "relied heavily on code in example applications that provided an implicit specification for reuse of the target class." The significance of examples encouraged researchers to develop example repositories to enable programmers to find examples easily [23, 37]. Unfortunately, the effectiveness of these repositories was limited by the retrieval mechanism which required too much (and too complex) input from programmers.

More recently, MAPO [38] and Strathcona [18] automatically retrieve examples from the structure of the program the programmer is writing. In a controlled experiment, participants using MAPO produced code with fewer bugs than participants in other conditions. This result is notable because it shows that API interventions can produce higher quality responses, not just more rapid responses.

The eMoose IDE plugin has proven similarly useful to developers using complex API specifications [9].The eMoose tool pushes directives—rules required to use a method correctly—to the method invocation site. The concrete rules that make up a protocol (e.g. one cannot call setDoInput on a connected URLConnection) are examples of directives. Dekel's evaluation of eMoose demonstrated significant programmer performance improvements during library-usage tasks (including one library with a protocol).

Unfortunately, examples and directives are labor intensive for API designers to produce. In large complex APIs it is often impossible to generate examples for every possible use case. Even after they are produced, it is hard to keep them in sync with the API as it changes, because there is no mechanism to enforce conformance. Examples can also serve as a crutch toward learning, and the most effective students learn to generate their own examples [6].

The design of Plaiddoc is inspired by all of the research discussed in this section. We modify Javadoc to produce an informationally equivalent documentation format aimed at facilitating speedier state search. Plaiddoc is generated from specifications whose conformance with code can be checked automatically. Plaiddoc specifications, like eMoose directives, are co-located with each method. The specifications themselves contain just the right state details so programmers can generate their own examples of correct API usage. The details of the Plaiddoc design are discussed in the next section.

3 Plaiddoc

To follow the rest of this paper, it is important to understand the design of Plaiddoc. To do so, it is necessary to first explain Javadoc. Javadoc is a tool for generating HTML documentation for Java programs. The documentation is generated from Java source code annotated with "doc comments" which contain both prose description and descriptive tags which tie the prose to specific program features. For example, a doc comment on a method will describe the method in general and then provide tags and associated comments for the parameters, the return value, and/or any exception the method throws.

The webpage generated by Javadoc for a class has six parts. The top and bottom contain navigation elements which allow the reader to quickly browse to related documentation. The class description appears below the navigation elements at the top of the page. It states the name of the class and links to superclasses and known subclasses. It then follows with an often long description which can include: the purpose of the class, how it is used, examples of use, class-level invariants, relationships to other classes, etc.

After the class description, the page includes four related elements: the field summary, method summary, field details, and method details. The field summary is a table containing the modifier, type, name, and short description of each public field sorted in alphabetical order. The method summary is extremely similar: it shows the modifier, return type, method name, type and name of all parameters, and short method description in alphabetical order. The field and method details show each field (or method) in the order they appear in the source file with the full description including historical information and any tags.

The Plaiddoc generated webpage maintains all of the look and feel of the Javadoc page. The fonts, colors, and visual layout are identical. However, the method summary section is restructured and extra information is added to the method details section. The full ResultSet page is available on the web.[1] The screenshot shows the method summary for the top-level Result state and the Open state.

As in Plaid, methods in the summary are organized by abstract state. In Javadoc, there is one table containing all of the methods of a class, while in Plaiddoc there is one table per abstract state. For example, the Disconnected state of URLConnection has a table containing all of the methods available in it, including setDoInput and connect.

One important rule we followed when designing Plaiddoc is that there is exactly one Plaiddoc page per Javadoc page. This rule ensures that the any observed differences between participants using Plaiddoc and Javadoc is a consequence of Plaiddoc's extra features and not the result of differences in page switching. There are two consequences of this rule: 1) All of the possible states of single Java class appear in the same Plaiddoc page.[2] 2) Multi-object protocols appear in multiple Plaiddoc pages. Six of the tasks in this study involve the Timer and TimerTask classes which impose a multi-object protocol. In these tasks, Javadoc participants were given two pages and Plaiddoc participants were given two pages.

An automatically generated diagram which shows all of the states of the class and where the particular state fits in, appears above each state table. The current state is bolded and italicized, while other states are displayed in the standard font. This diagram is *primitive*; it does not contain extensive capabilities like hyperlinks from state names to state tables, collapsing/expanding children, transition arrows, or even a nice graphical look. The diagram is primitive for

[1] `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/PlaiddocResultSet.html`

[2] e.g. The "Open" and "Closed" states of ResultSet appear on a single page.

three reasons: 1) Plaiddoc was designed for this experiment, and was therefore not polished for use outside the laboratory. 2) More capabilities gives participants more potential paths to solve tasks and thus introduces variation into the study. 3) If one adds features it is harder to understand which particular features are important or unimportant. Plaiddoc was designed with the minimum set of features we believed would be an effective group.

The Plaiddoc page also contains two new columns in the method details table. These columns are state preconditions and postconditions. The only valid predicates are state names, state names with a parameter, or combination of the two separated by the AND or OR logical operators. For example, "Disconnected," "Scheduled task," and "Updatable AND Scrollable" are valid preconditions or postconditions but "value > 0" is not. The same information is added to the method summary. The state to which a method belongs is an implicit precondition for that method. For example, the close method lists no preconditions, but since it belongs to the Open state, the ResultSet must be in the Open state to call the close method.

To generate a Plaiddoc class page, the Plaiddoc tool requires three inputs: the class's Javadoc page, a JSON file specifying the state relationships of the class, and a JSON file containing preconditions and postconditions for each method and mapping methods to states. Sample JSON files are available on the web.[3]

The JSON files are very simple. The state file must contain a single object whose fields are states, each of which must contain either an "or-children" or "and-children" field. These "children" fields are arrays containing state names, which in turn must be defined in the same file. The methods file must contain an array of method objects which contain four fields: "name" (including parameter types to distinguish statically overloaded methods), "state" (which must map to a state defined in the state file), "pre" for preconditions, and "post" for postconditions.

It is important to map the features of Plaiddoc just described to concepts, in order to understand the implications of the experiment described here on other research (e.g. the Plaid language itself). Plaiddoc organizes methods by state instead of by class, by separating the method summary table by state. Plaiddoc makes state transitions explicit when state postconditions differ from preconditions. The Plaiddoc preconditions and postconditions make use of state-based type specifications. Finally, rich state relationships are displayed to programmers at the top of each method table. See e.g. the "State relationships" box.

4 State search categories

As we mentioned in Section 1, an earlier two-part qualitative study of the barriers programmers face when using APIs with protocols feeds directly into the methodology of the study in this paper [31, ch. 3]. In the first part of that study, we mined the popular developer forum StackOverflow for problems developers

[3] `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/Car_States.json` and `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/Car_Methods.json`

have using APIs with protocols. In the second part, they performed a think-aloud observational study of professional programmers in which the programers worked through exactly the problems uncovered in the first part.

In this second part, they analyzed each task, by assigning task time to participant questions or comments and performing open coding on the transcript. This analysis showed that programmers in spent 71% of their total time answering instances of four question categories. We list here each general category followed by two specific instances of that category drawn from the study transcripts:

A What abstract state is an object in?
– "Is the TimerTask scheduled?"
– "Is [the ResultSet] x scannable?"
B What are the capabilities of an object in state X?
– "Can I schedule a scheduled TimerTask?"
– "What can I do on the insert row?"
C In what state(s) can I do operation Z?
– "When can I call doInput?"
– "Which ResultSets can I update?"
D How do I transition from state X to state Y?
– "How do I get off the insert row to the current row?"
– "Which method schedules the TimerTask?'

These search problems are all specific to protocols, and therefore the protocol tasks are dominated by state search. Most of the tasks performed by participants in this study are instances of these general categories.

5 Methodology

The experimental evaluation of Plaiddoc uses a standard two by two between-subjects design, with five participants in each of the four conditions. The experiment compares Plaiddoc to a Javadoc control and presents two task orderings to measure learning effects. The recruitment, training, experimental design, tasks, and post-experiment interview are presented in the following sections. All of the study materials can be found in Appendix C [31].

5.1 Recruitment

All 20 participants were recruited on the Carnegie Mellon campus. Half of the participants responded to posters displayed in the engineering and computer science buildings. The other half were solicited in-person in a hallway outside classrooms which typically contain technical classes. Participants were screened for Java or C# knowledge and experience with standard API documentation. Participants were paid $10 for 30-60 minutes of their time. The 20 participants that made it past the screening all completed the study.

Twelve of the participants were undergraduate students, all of whom were majoring in computer science, electrical and computer engineering, or information

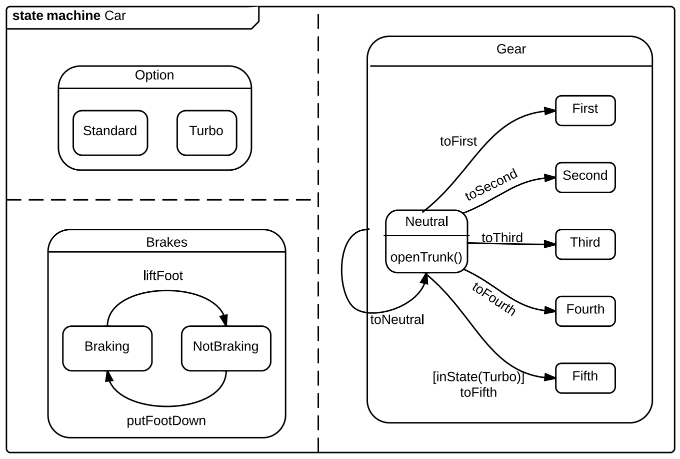

Fig. 1. Car state machine used for participant training.

systems. The other eight were masters students in information systems or computer engineering programs. Eleven students had no professional programming experience outside summer internships, five students had one year of full-time professional experience, and four had more than one year of experience.

5.2 Training

After signing consent forms, participants were given approximately 10 minutes of training. Every participant, regardless of experimental condition, received exactly the same training. The training was read from a script to help ensure uniformity.

All participants were familiar with Javadoc, but the training included an explanation of both Javadoc and Plaiddoc to ensure baseline knowledge in both formats. The goal of this study is to compare the impact of the documentation formats on state search tasks, not the impact of training. Therefore, we kept training consistent to avoid a confounding factor. All of the state concepts are first taught via UML state machines, then Javadoc, then Plaiddoc.

The training materials introduce participants to the basic concepts of object protocols and to the documentation formats used in the study. The training makes concepts concrete using a Car API we constructed for the purpose. Regarding protocols, participants learn:

- that methods are available in some states and not others
- that some methods transition objects between states
- that states can be hierarchical
- that child states can be either or-children or and-children

These concepts were reinforced by asking participants simple, scripted questions about the Car API. The questions were designed to be answerable very quickly by participants. We created a UML state machine (shown in Figure 1), Javadoc documentation, and Plaiddoc documentation for the Car API and these were printed and handed to participants.

The top-level state for Car objects (named "Car") has three and-children, each of which has two or more or-children: *gear* to represent the car's manual transmission, *brakes* to represent whether the car is braking or not, and *option* to represent whether the car has the "turbo" option or not. We used these states to introduce state hierarchy, or-states, and and-states. We introduced transitions via brakes. One can transition to the "Braking" state from the "NotBraking" state by calling the "putFootDown" method. The `openTrunk` method, which does not change the gear state, introduces state-dependent methods. In the example, like in many real-world cars, one can only open the trunk when the car is in the neutral gear.

Like all and-children, the car's three substates are independent, in the sense that changing the gear state has no effect on the braking or option states. However, one unique wrinkle in the example is that the turbo state enables a fifth gear substate of gear that is not available otherwise. The `toFifth` method has two preconditions — the car must be in the neutral gear and it must have the turbo option. In the study tasks discussed later, some of the ResultSet methods also have multiple preconditions.

5.3 Experimental Setup

Participants were asked 21 questions about three Java APIs: 1) Six questions about `java.util.Timer` and `java.util.TimerTask`. We refer to these questions as the Timer questions throughout the rest of this paper. 2) Ten questions about `java.sql.ResultSet`. 3) Five questions about `java.net.URLConnection`. The experimenter read each question aloud and handed the participant a piece of paper with the same question written on it.

Participants were seated in front of a computer, and asked to answer the question by looking at documentation on the computer screen. The experimenter opened the documentation for the participant in a browser window. Both the Javadoc and Plaiddoc documentation were opened from the local file system to present a consistent URL and to prevent network-related problems. The computer screen and audio (speech) were recorded with Camtasia.

Half of the participants were shown standard Javadoc documentation for all questions and half Plaiddoc documentation. Participants were allowed to make use of the browser's text search (i.e. Control-F). However, they were not allowed to use internet resources (e.g. Google, StackOverflow).

We chose a between-subjects design to control for cross-task contamination. Many software engineering studies use within-subjects designs to reduce the noise from individual variability. We guessed based on pilot data that individual variability in our study would be relatively low and we therefore opted for the cleaner between-subjects design. As we will see in §6, the study was sufficiently sensitive to distinguish between conditions so our guess turned out to be accurate.

Questions were asked in batches — all of the questions related to a particular API were asked without interruption from questions about another API. Within each batch, each question was asked in the same order to every participant. However, half of the participants were asked the Timer batch first and half were asked the UrlConnection batch first. The ResultSet batch always appeared second and the remaining batch appeared third. We wanted the Timer and URLConnection batches to each appear last so we could measure the learning effects on those batches. All other ordering was uniform across conditions to avoid unnecessary confounding factors.

The study had a total of four between-subjects conditions: Plaiddoc with Timer first (condition #1), Plaiddoc with URLConnection first (condition #2), Javadoc with Timer first (condition #3), and Javadoc with URLConnection first (condition #4). Participants were assigned to conditions based on the order they appeared in the study. The nth participant was assigned to condition #n modulo 4. Using commonly accepted practice, participants were assigned to conditions pseudorandomly, in the order they arrived. Therefore, there were exactly five participants in each condition.

5.4 Tasks

The 21 questions asked of the participants are shown in Table 1. Sixteen of the questions were instances of the four categories of state search enumerated in §**??**. Since these questions are state specific, we refer to them as the state questions. The remaining five questions were non-state questions, which were designed to be just as easy or easier with Javadoc than Plaiddoc. These questions were not about states or protocols, and we therefore refer to them as the non-state questions.

We selected the state questions with a three-phase process. First, we generated all of the instances of the general categories we could think of for each API. Second, since we did not want the answer or the process of answering one question to affect others, we removed questions which were not independent. Some additional non-independent questions were removed during piloting. Third, we pruned the ResultSet questions to include two instances of each question category by random selection. The study was too long with the full set of ResultSet questions.

The final question set includes three instances of A) "What abstract state is an object in?", five instances of B) "What are the capabilities of an object in state X?", four instances of C)"In what state(s) can I do operation Z?',' and four instances of D) "How do I transition from state X to state Y?" Participants in all conditions were given a glossary listing all of the states of the API in question with a short description of each. Participants were instructed to answer questions in categories A and C with the name of a state from the glossary. In other words, these questions were multiple choice.

The names of states in the glossary matched those in Plaiddoc. The names themselves were taken from the Javadoc as much as possible. We did not want

Table 1. Category, identifier and question text for all of the questions asked of participants in the main part of the study. Questions with identifiers beginning with T involved `java.util.Timer` and `java.util.TimerTask`, R involved `java.sql.ResultSet`, and U involved `java.net.URLConnection`.

Cat.	ID	Question text
T	T-1	How do I transition a Timer Task from the Virgin state to the Scheduled state?
N	T-2	What is the effect of calling the purge method on the behavior of the Timer?
C	T-3	What methods can I call on a Scheduled TimerTask?
N	T-4	What is the difference between schedule(TimerTask task, long delay, long period) and scheduleAtFixedRate(TimerTask task, long delay, long period)?
O	T-5	What state does a TimerTask need to be in to call scheduledExecutionTime?
C	T-6	Can I schedule a TimerTask that has already been scheduled?
N	R-1	How is a ResultSet instance created?
C	R-2	Can I call the getArray method when the cursor is on the insert row?
O	R-3	What state does the ResultSet need to be in to call the wasNull method?
T	R-4	How do I transition a ResultSet object from the ForwardOnly to the Scrollable State?
O	R-5	Which states does the ResultSet need to be in to call the updateInt method?
A	R-6	What state is the ResultSet object if a call to the next method returns false?
T	R-7	How do I transition a ResultSet object from the CurrentRow to the InsertRow state?
N	R-8	Why does getMetadata take no arguments and getArray take a int columnIndex or String columnLabel as an argument?
C	R-9	Can I call the isLast method on a forward only ResultSet?
A	R-10	What states could the ResultSet object in when a call to the next method throws a java.sql.SQLException because it is in the ResultSet is in the wrong state?
A	U-1	What state is the URLConnection in after successfully calling the getContent method?
C	U-2	If the URLConnection is in the connected state can I call the setDoInput method?
N	U-3	How do I create a URLConnection instance?
O	U-4	What state does the URLConnection need to be in to call the getInputStream method?
T	U-5	What method transitions the URLConnection from the Connected to the Disconnected state?

Category definitions

A	Instance of the "What abstract state is an object in?" question category.
C	Instance of the "What are the capabilities of an object in state X?" question category.
N	Instance of the non-state question category.
T	Instance of the "How do I transition from state X to state Y?" question category.
O	Instance of the "In what state(s) can I do operation Z?" question category.

to disadvantage Javadoc unnecessarily, so we tried to make it as easy as possible for participants to perform the mapping from the prose description in the Javadoc to the state names in the glossary. In two cases there was no obvious name to give the state from the Javadoc. First, we called a URLConnection that has not yet connected "Disconnected," which is a word that appears neither in the Javadoc nor the Java source code. Second, we called a TimerTask that is unscheduled, "Virgin" even though this word never appears in the Javadoc. In this case we borrowed the word from the implementation code—the state of a TimerTask is encoded with an integer, and the integer constant used for an unscheduled TimerTask is called VIRGIN. Finally, we wrote all of the descriptions to succinctly explain the meaning of the state name.

All of the non-state questions require understanding a non-state detail of the API or comparing two details. Since the Plaiddoc API documentation is larger than the Javadoc documentation one might expect that it would be slightly easier to answer these questions with Javadoc. Two of the non-state question are instances of "how do I create an instance of class X?", two ask participants to compare two methods (in one case the methods were in different states), and one asks participants to understand non-state details of the behavior of an individual method.

Participants were instructed to "find the answer to each question in the documentation and tell the experimenter the answer as soon as you have found it." Whenever a participant answered a question for the first time, the experimenter asked,"is that your final answer?" Participants were limited to ten minutes per task. The experiment proceeded to the next task whenever a participant answered a question and confirmed it or the time limit was reached. Participants were not told whether their answer was correct and the experiment proceeded regardless of answer correctness.

5.5 Post-experiment interview

After completing the experiment participants were asked four questions to see how well they could map the state concepts we trained them about before the study (e.g. and-states, or-states, state hierarchy, impact of transitions on and-states) to the particular APIs they saw in the study. For example, we asked "What is an example of two ResultSet and-states?" Participants were also asked to rate their affinity to the documentation they used, and if they used Plaiddoc to compare Plaiddoc to Javadoc on a five point Likert scale. Then they were asked "Which documentation format that you learned about before the study—Javadoc, Plaiddoc, or UML state diagram—do you think would have been most helpful to complete this study?" Finally, some individuals were also asked additional questions about their task performance at the experimenter's discretion.

6 Results

In this section, we discuss the study results and try to give the best evidence to answer the research questions presented in the introduction. We first compare the task completion performance of Plaiddoc and Javadoc participants. Then we compare the correctness of these responses provided by those same groups. We follow with an evaluation of the learning effects of performing study tasks. Finally we discuss the post-study interview and pilot results. Raw timing and correctness data is available on the web.[4]

6.1 Task Completion Time

In this subsection we discuss the results related to the task completion time output variable. This output variable addresses RQ1 and RQ2 (Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc? and Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?) by comparing task completion times across conditions.

To determine completion time we analyzed the video and marked when we finished reading the task question and when the participant confirmed his or her "final answer." The difference between these two marks was noted in the task completion time.

The ten-minute task time limit was reached by many participants on question R-4, but never on any other question. In fact, only two participants exceeded five minutes while answering any other question, and they did so for only one question each. Timeouts are not directly comparable to other timing data, and therefore we evaluate question R-4 separately, and in detail, in §6.2. This subsection does not include data from question R-4.

The total completion time for each of the Plaiddoc and Javadoc participants on state questions is visualized by the box plot in Figure 2(a), and for non-state question in Figure 2(b). A two-factor fixed-effects ANOVA revealed no significant interaction between documentation type and task ordering (p=0.25) on total task completion time. Therefore, we compare all 10 Plaiddoc participants against their 10 Javadoc counterparts.

The mean total completion time of all state search tasks was 10.3 minutes in the Plaiddoc condition, and 22.4 minutes in the Javadoc condition (2.17x difference). An independent samples two-tailed t-test revealed that the difference is statistically significant ($p < 0.001$). The difference between the means was 12.1 minutes, and 95-percent confidence interval was 6.38 to 17.8 minutes.

The mean completion time of non-state tasks was 5.77 minutes in the Plaiddoc condition, and 5.95 minutes in the Javadoc condition. Unsurprisingly, this difference is not statistically significant (p=0.802). The 95-percent confidence interval of the difference is -1.32 to 1.68 minutes.

The four state search categories can be subdivided into two categories. In two of the search categories, a participant begins his or her search at a state and tries to find a method.[5] In the other two search categories the participant starts

[4] `http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/RawPlaiddocStudyData.pdf`

[5] What are the capabilities of an object in state X? How do I transition from state X to state Y?

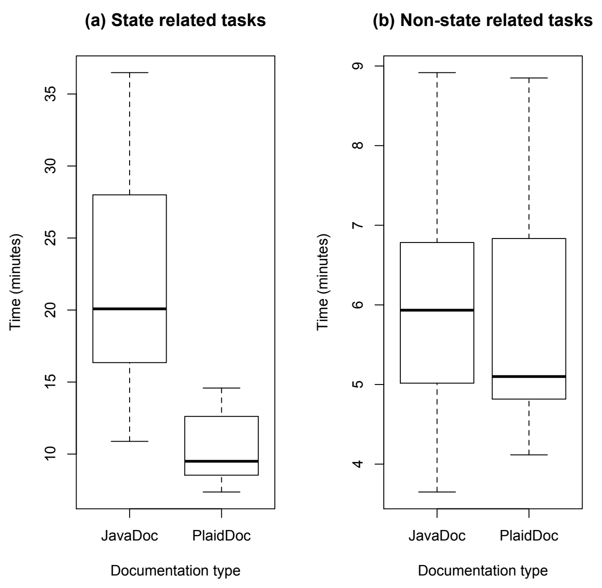

Fig. 2. Box plot comparing the completion time of Javadoc and Plaiddoc participants.

at a method or other detail (e.g. exception, instance creation), and tries to find a state.[6] Since methods are organized in Plaiddoc by state one would expect that Plaiddoc would improve performance primarily for searches that proceed from a state to a method. This hypothesis turns out to be correct — Plaiddoc outperformed Javadoc in these categories by 2.41x. However, one might expect that Plaiddoc would not be helpful in the method first categories, but Plaiddoc outperformed Javadoc by 1.87x in these categories. Therefore, Plaiddoc appears to be more helpful for state-first search than method-first search. We performed two factor, fixed-effects ANOVA in which the two factors are documentation type and search type and the output variable is time. The interaction term between documentation type and search type is only marginally significant (p=0.089).

Demographics. We did not balance participants in conditions by any demographic factor. By random chance, six of nine students with experience and three of four with more than one year of experience were assigned to the Javadoc conditions. However, experience had no significant impact on the timing results. A two-factor ANOVA where the two factors were experience and documentation type showed no significant effects from experience (F=.058, df=1, p=.813) or the experience by documentation type interaction term (F=1.34, df=1, p=.719).

[6] What abstract state is an object in? In what state(s) can I do operation Z?

Table 2. Correctness results for each participant on the 16 state search questions.

	Paricipant #																				Total	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Pdoc	Jdoc
DocType	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	J
Correct	15	15	14	16	15	16	15	14	15	15	14	14	15	15	16	16	15	15	11	13	151	143
Incorrect	1	0	2	0	0	0	1	1	0	0	2	2	1	0	0	0	0	0	5	2	2	15
Timed-out	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	0	1	7	2

Feature comparison discussion. Every participant used text-search (i.e. CTRL-F in the browser window) to find method names. They then used the location in a state box, pre-conditions, post-conditions, and state relationship diagrams to answer the question efficiently. Plaiddoc is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. The difference in relative performance between the state categories allows us to (very roughly) compare the benefits of state organization to the other three features. Since the method based search does not benefit from the state-based organization, all of the performance differences observed in the method based search tasks are likely to derive from explicit state transitions, state-based type specifications, and rich state relationships. The extra performance of the state based search is likely to derive from the state-based organization. We do not think it's possible to separate the benefits of the embedded state diagram from the preconditions and postconditions. In one early pilot we did not include the state diagram and the participant struggled to answer questions that required knowledge of state relationships. Similarly, a state diagram without detailed information about the requirements and impact of method calls would likely not be effective.

6.2 Correctness

Almost half of the participants provided at least one wrong "final" answer to a state-search question. Among the 320 total answers provided to the 16 state search questions 294 were correct, 17 incorrect, and nine were not provided because the question timed out. In this subsection, we compare the correctness of Plaiddoc answers to Javadoc answers (RQ3). The number of right, wrong, and timed-out answers for each participant are shown in Table 2.

Only two of the 17 wrong answers were provided by Plaiddoc participants. Plaiddoc participants answered 98.75% of the questions correctly, and Javadoc participants answered 90.5% correctly. The odds ratio in the sample is 7.92.[7] We analyzed the contingency table of Javadoc vs. Plaiddoc and Correct vs. Incorrect using a two-tailed Fisher's exact test. The contingency table is shown in Table 2 in the rows labeled "Correct" and "Incorrect" and the columns labeled "Pdoc" and "Jdoc". The test revealed that the difference is very significant (p=0.002). The 95-percent confidence interval of the odds ratio is 1.78 to 72.1.

[7] The odds ratio is a standard metric for quantifying association between two properties. In our example, it is the ratio of the odds of being correct when using Plaiddoc to the odds of being correct when using Javadoc.

Incorrect responses. All of the wrong answers and time-outs were provided to just five of the 16 state questions. No wrong answers were provided to any of the non-state questions. It is worth discussing the content of the wrong answers to provide insight into the types of problems programmers face when answering state-related questions.

In response to question T-3, a Plaiddoc participant (#19) incorrectly suggested that none of the TimerTask methods could be called on a scheduled TimerTask because "the methods are called by the Timer." This participant correctly noted the main mode of usage, but incorrectly assumed this was the exclusive mode of usage.

In response to question T-5, three[8] Javadoc participants incorrectly suggested that TimerTask scheduledExecutionTime can be called in any state when in fact it can only be called in the executed state. Three of these wrong participants noted correctly that scheduledExecutionTime does not throw an exception. Unfortunately, not every protocol violation results in an exception, a fact that was noted in pre-test training.[9] In this case, the protocol is documented in the description of the return value, which is described as "undefined if the task has yet to commence its first execution." In the post-experiment interview all three incorrect participants said that they did not notice this return value description.

In response to T-6, two Javadoc participants incorrectly replied that one can schedule an already-scheduled TimerTask. Participant #19 answered very quickly (15 seconds) without thoroughly examining the documentation. Participant #8 read aloud from the documentation, noting that the method throws an IllegalStateException "if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated." However, #8 somehow skipped "scheduled or" while reading.

Three Javadoc participants and one Plaiddoc participant incorrectly answered U-5. The question asks, "What method transitions the URLConnection from the Connected to the Disconnected state?" There is no such method, as 16 participants correctly noted. The three incorrect Javadoc participants suggested one could transition the URLConnection to the Disconnected state by calling its setConnectionTimeout method with 0 as the timeout value argument. This method "sets a timeout value, to be used when opening a communications link to the resource referenced by this URLConnection. If the timeout expires before the connection can be established, a java.net.SocketTimeOutException is raised." Therefore, setConnectionTimeout has no impact at all on a URLConnection instance that has already connected. Participant #1, a Plaiddoc participant, incorrectly answered that the non-existent "disconnect" method could be used to transition the URLConnection. This was the last question that participant

[8] Participant #19 also answered T-5 incorrectly because, as in question T-3, #19 thought all TimerTask "methods are called by the Timer" including scheduledAtFixedRate.

[9] The openTrunk method's protocol is documented by its description of the return value Javadoc training materials.

#1 answered, so perhaps #1 was ready to leave and so didn't investigate this question thoroughly.

Finally, R-4 produced the most varied responses. The question asks the participant to transition a ResultSet object from the ForwardOnly to the Scrollable state. However, no transition is possible since ForwardOnly and Scrollable are *type qualifiers* and therefore are permanent after instance creation. Seven Plaiddoc and two Javadoc participants never answered this question because they timed out. One Plaiddoc and five Javadoc participants answered the question incorrectly. Many of the timed-out Plaiddoc participants considered but then ultimately rejected the incorrect answers provided by the Javadoc respondents. This suggests that the specifications provided by Plaiddoc participants can provide confidence that an answer is *incorrect*. The Plaiddoc participants likely traded no-answers for incorrect answers.

Four Javadoc participants incorrectly answered that the setFetchDirection method will transition a ResultSet object from the ForwardOnly to the Scrollable state. Unfortunately, this method does no such thing, instead it "gives a hint as to the direction in which the rows in this ResultSet object will be processed." These four participants did skim the description, but it seems that they relied primarily on the method name to make their determination.

One Javadoc and one Plaiddoc participant noticed the following sentences in the class description: "A default ResultSet object is not updatable and has a cursor that moves forward only ... It is possible to produce ResultSet objects that are scrollable." which is immediately followed by a code example in which the createStatement method is called on TYPE_SCROLL_INSENSITIVE as an argument on a *connection* instance. Upon reading this, both participants immediately answered that the createStatement method should be called on a *ResultSet* instance. The Plaiddoc participant even suggested that the createStatement was missing from the method details list because "Plaiddoc is just a prototype."

Questions U-5 and R-4 both ask participants to find a method that does not exist. These questions, like all state-search questions in the study, are derived from the questions participants asked in the observational study discussed in Sunshine [31, ch.3]. However, participants in empirical studies are well-known to be compliant to experimenter demands. Therefore, some may therefore consider them to be "trick" questions. If these questions are excluded, then Plaiddoc participants answered 140 state-search questions correctly (100%) and 0 incorrectly while Javadoc participants answered 133 correctly (95%) and 7 incorrectly. A two-tailed Fisher's exact test of this contingency table is statistically significant (p=0.014). Since Plaiddoc participants in this sample answered every question correctly, the odds ratio is infinite. The 95-percent confidence interval of the odds ratio is 1.48 (the corresponding value is 1.78 when including every state-search question) to infinity (7.92 when including state-search question). Therefore, Plaiddoc participants were significantly more likely to respond correctly than Javadoc participants even when excluding "trick" questions.

Discussion. Three themes emerge from the incorrect and timed-out answers provided by participants. First, all of the time-outs occurred in question R-4 when participants were asked to find a non-existent method to transition between two states. Therefore, to answer this question correctly, participants needed to prove the absence of something to themselves.[10] Some participants felt the need to perform a brute force search of the method documentation to ensure that no methods were available that perfumed the transition. Of particular note, Plaiddoc participants didn't seem to trust that the ForwardOnly section of the Plaiddoc contained all of the potential methods.

It is also worth noting that question U-5 is in the same category but resulted in no time-outs. One possible explanation is that the ResultSet interface is much larger than the the URLConnection class,so it is easier to be confident that no such method exists. In addition, participants seemed to intuit that the URL-Connection transition is missing, but not intuit that the ResultSet transition is missing.

Second, the questions required the participants to digest a lot of text. Participants commonly relied on heuristics and skimming to answer questions quickly. For example, the five Javadoc participants who answered R-4 with setFetchDirection matched the method name to the task and quickly confirmed the match in the description, but did not fully digest the description text. The participant who missed the word "scheduled" in the exception details was being similarly hasty. This phenomenon may partially explain why Plaiddoc participants were so much quicker than Javadoc participants, as we saw in §6.1. Plaiddoc presents a natural heuristic to participants — when examining a method, look first at the state it is defined in, then at its preconditions and postconditions.

Third, participants were tripped up by non-normal modes of use. We saw that participant #19 thought only the Timer could call TimerTask methods because that is the normal mode of use. Similarly, most protocol violations throw exceptions and are documented in the method or exception descriptions. However, scheduledExecutionTime somewhat abnormally documents the protocols in the return value description which confused three participants. Finally, abstract states normally map well to the primitive state of object instances. However, a URLConnection that has been disconnected from the remote resource is not in the Disconnected abstract state, as expected by three participants.

6.3 Learning

To answer RQ4, which asks whether state search performance improves with practice, we alternated the order that question batches were asked of participants. As we describe in §5.3, half of the participants first received URLConnection questions and half first received Timer questions. The output variable we discuss in this section is the ratio of total Timer batch completion time to total URLConnection batch completion time (the "T/U ratio"). If learning occurs,

[10] In [31, ch.3] many forum questioners had similar problems with missing state transitions.

Table 3. Analysis of observed variance of T/U Ratio. The fixed-effects sources of variation considered are documentation type and batch order.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
DocType	1	0.06695	0.06695	0.4560	0.50914
BatchOrder	1	0.96519	0.96519	6.5737	0.02081
DocType:BatchOrder	1	0.51496	0.51496	3.5073	0.07949

then the T/U ratio should be larger for participants who performed the Timer batch first than for those who performed the URLConnection batch first.

In the Javadoc condition, the mean T/U ratio of the Timer first sub-condition is 1.07 and .948 in the UrlConnection first sub-condition. This difference is not statistically significant (p=0.695). On the other hand, in the Plaiddoc condition the mean T/U ratio of the Timer first sub-condition is 1.50 and 0.743 in the UrlConnection first sub-condition. An independent samples two-tailed t-test shows that this difference is statistically significant (p=0.003).

We performed a two factor, fixed-effects ANOVA in which the two factors are documentation type and batch order and the output variable is the T/U ratio. The results are show in Table 3. This ANOVA reveals that there is a marginally significant interaction between documentation type and batch ordering (p=0.079). This should be interpreted as weak evidence that task-completion speed improved more for Plaiddoc participants than for Javadoc participants. However, more data is needed to know for sure.

Discussion. The Plaiddoc participants performance improved significantly during the study, which is perhaps unsurprising since Plaiddoc was new to all of the participants. We would like to say with confidence that state-search performance of programmers using Plaiddoc would improve over time relative to programmers using Javadoc. However, the learning observed in the Plaiddoc condition was not significantly stronger than the learning observed in the Javadoc condition.

6.4 State concept mapping

To investigate RQ5, we asked four questions to map the concepts they learned about in training to the Timer, TimerTask, ResultSet, and URLConnection. Plaiddoc participants responded correctly 23 of 40 times, while Javadoc participants answered correctly 25 times. This difference is not statistically significant.

Discussion. We hypothesized that Plaiddoc participant would be better at mapping API specifics to general state concepts. We thought this because Plaiddoc makes many state concepts more salient. There is no evidence for this hypothesis in the data. Javadoc participants spent much more total time with the documentation and they read much more of the detailed prose contained inside the documentation. Perhaps this extra time and detail compensated for the state salience of Plaiddoc.

We told all of the participants that timed out while trying to find a method to transition the ResultSet from ForwardOnly to the Scrollable state, that the method did not exist. We asked if they had any ideas about how to better represent missing state transitions. Most didn't give any suggestion, but one suggested that methods that perform state transitions should be separated from other methods so they're easier to find. This suggestion is worthy of further investigation.

6.5 Participant preference

In the post-experiment interview we also gauged participant preferences. Nine of ten Plaiddoc participants said that a different documentation format would have been more helpful in performing the study. Seven selected UML state diagrams and two selected Javadoc. The Javadoc participants also primarily selected UML State diagrams (five of ten), followed by Javadoc (3), and Plaiddoc (2).

Discussion. The results in this study show that Plaiddoc participants outperformed Javadoc participants. Therefore participant preferences does not match the measured outcome. Why do so many Plaiddoc participants prefer another documentation format? The simplest explanation is that Plaiddoc is unfamiliar, while Javadoc is familiar. In addition, one participant in the Plaiddoc condition who preferred Javadoc explained that he "felt lost" while using Plaiddoc. A Plaiddoc page is divided into many more subsections (one for each state) than a Javadoc page. Improved visual cues indicating the which state is being viewed might alleviate this problem. Another possible reason, is that the Plaiddoc state diagram is produced in ASCII and therefore looks old and amateurish. The state diagram does not match well with the modern look of the rest of the page. Regardless of the reason for the preference, this study's results are a cautionary tale for researchers who rely only on user preferences to evaluate tools.

7 Threats to Validity

In this section we discuss threats to validity of our causal claims. We divide this section using the canonical categories of validity: construct validity, internal validity, and external validity.

7.1 Construct validity

We trained all participants equally, including training of Javadoc participants to use Plaiddoc. There is some risk in this design that Javadoc participants will be disappointed that they did not get to use Plaiddoc. They were familiar with Javadoc so they may have preferred to try something new. Therefore, Javadoc participants may have performed worse because they experienced what Shadish [27, p. 80] calls "resentful demoralization." Two facts suggest that demoralization had at most a small effect on the results: First, only two of 10

Javadoc participants said they would have preferred to use Plaiddoc in the post-experiment interview. Second, both Javadoc and Plaiddoc are documentation formats and neither is particularly exciting. The classic examples in which "resentful demoralization" was measurable include much more severe differences between the control group and the experimental group. Fetterman [12] describes an experiment evaluating a job-training program in which the control group includes participants who were denied access to the training program. Walther [35] compared an experimental group that is paid a substantially higher participation reward to a control group paid much less. We would not expect to see anywhere near as much demoralization in our study as in these studies, even for participants who would have preferred to use Plaiddoc.

Although participants were never told explicitly, it is likely participants realized that Plaiddoc was our design. Therefore, Plaiddoc participants may have performed better and Javadoc participants worse because of "experimenter expectancies" [25, p. 224]. In other words, the very fact that we expected Plaiddoc to outperform Javadoc *and* the participants could possibly infer this expectation, may have impacted in the result in the direction we expected.

7.2 Internal validity

The focus of this study's design is internal validity. Participants were randomly assigned, participants were isolated from outside events in equivalent settings, we used a between-subjects design, and there was no attrition during the study. All that being said, one threat to internal validity is worth mentioning. Participants were assigned to conditions randomly, but it could be that the participants in the Plaiddoc group were better equipped to answer the questions in the study. We discussed the distribution of programming experience in §6.1 and showed that it did not seem to have an effect on outcomes. However, it could be the groups differ along another dimension—for example, programming skill, experience with protocols, intelligence—that we did not measure and this impacted the results.

7.3 External Validity

Our earlier qualitative studies and the experiment discussed here have opposing strengths and weaknesses. The qualitative studies emphasize external validity with realistic tasks and professional participants, but cannot be used to draw conclusions about causal relationships. The experiment in this paper focuses on internal validity with a carefully controlled experimental design that allows strong causal conclusions. However, the external validity of the experiment is enhanced because participants performed tasks in which they were required to tackle protocol programming barriers observed in the qualitative studies. Therefore, the experimental results are likely to translate to real-world problems and the processes that programmers use to solve them. All that being said, the threats to external validity in those earlier studies extend into this study [31, §3.4].

The state search tasks are connected to our qualitative results—they use the same APIs that were problematic for Stack Overflow questioners and they are instances of the state search categories that were observed repeatedly in the observational study. However, the non-state search tasks did not come from developer forums or any other real-world programming resource. Instead they were designed to simply *not* make use of Plaiddoc's novel state features. In our results, Plaiddoc participants did not perform worse on these tasks than Javadoc participants. However, it could be that there are other important categories of tasks for which Javadoc is better than Plaiddoc.

Another noteworthy external validity concern in the experiment here has to do with the student population studied. None of the participants seem to have struggled with the concept of preconditions and postconditions which are used heavily by Plaiddoc. This may be because the concept as used in the study is simple, but it may also be that the Carnegie Mellon student population we studied is especially exposed to formal methods. The very first course in the Carnegie Mellon undergraduate computer science sequence teaches students to verify imperative programs with Hoare-style contracts.

8 Type annotations as documentation

Many research groups have developed specialized type-based annotation systems for particular domains. Prominent examples include information flow [26], thread usage policies [33], and application partitioning [7]. In the vast majority of these systems, including all of the examples just cited, the primary benefit of the annotation systems touted by their creators is either verification or automated code generation. The preconditions and postconditions that appear next to methods in Plaiddoc are essentially state-based type annotations. Therefore, this study provides indirect evidence that type based annotations can have benefits as documentation.

In the last few years, there have been a flurry of studies comparing the benefits of static and dynamic types [16, 29, 17]. This research suggests that dynamic types have an advantage for small, greenfield tasks, while static types have an advantage for larger, maintenance tasks.

The most closely related study [22], evaluated the benefits of type annotations in undocumented software. The results were mixed—types were significantly helpful in some tasks, and significantly harmful in others. One possible interpretation of the results is that types were helpful in tasks that were more complex (involved more classes) and harmful otherwise. Our results provide a clearer picture — Plaiddoc provided benefits in every state-search category. In their study, programmers performed programming tasks using two "structurally identical," synthetic, undocumented APIs. In our study, programmers answered search questions with well-documented real-world APIs. One important consequence of these differences, is that our study evaluates types *only* for their documentation purpose, while theirs evaluates the collective value of both static-checking and types as documentation.

9 Conclusion

In this study we demonstrate the effectiveness of Plaiddoc documentation relative to Javadoc documentation in answering state-related questions. The barrier to entry for using the Plaiddoc tool are minimal—only 1-3 annotations are required per method. We annotated all three APIs in less than one day of work. The main barrier to using Plaiddoc in production is training programmers to consume the documentation effectively. Untrained participants in pilot studies were not able to use Plaiddoc effectively. Even basic protocol concepts were foreign to our participants before training. That said, the training we provided was very quick and required no specialized knowledge. Regardless, it seems clear that any mainstream language that adopts first-class state constructs should also adopt a Plaiddoc like documentation structure. More generally, our study shows that state-based type annotations provide documentation-related benefits even for well-documented code. Thus, our results open the door to future work investigating the documentation-related productivity benefits of type-like annotations in a broad range of domains.

Acknowledgements. This work was supported by supported by the U.S. National Science Foundation under grants #CCF-1116907 and #IIS-1111750. National Security Agency lablet contract #H98230-14-C-0140, and the Air Force Research Laboratory.

References

1. N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP 2011 – Object-Oriented Programming*, pages 2–26. Springer Berlin Heidelberg, 2011.
2. N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 211–221, New York, NY, USA, 2011. ACM.
3. K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag.
4. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.
5. P. Chandler and J. Sweller. Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8(4):293–332, 1991.
6. M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145 – 182, 1989.
7. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM*

SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM.

8. G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 381–390, New York, NY, USA, 2011. ACM.
9. U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, 2009.
10. M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.
11. B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 302–312, 2007.
12. D. M. Fetterman. Ibsen's baths: Reactivity and insensitivity. (a misapplication of the treatment-control design in a national evaluation). *Educational Evaluation and Policy Analysis*, 4(3):261–279, 1982.
13. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.
14. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.
15. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.
16. S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.
17. S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, pages 1–48, 2013.
18. R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 237–240, New York, NY, USA, 2005. ACM.
19. C. N. Jaspan. *Proper Plugin Protocols*. PhD thesis, Carnegie Mellon University, December 2011. Technical Report: CMU-ISR-11-116.
20. B. E. John and D. E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4):320–351, Dec. 1996.
21. J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65 – 100, 1987.
22. C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 683–702. ACM, 2012.

23. L. R. Neal. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, pages 63–68, New York, NY, USA, 1989. ACM.
24. M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.
25. R. Rosenthal and R. L. Rosnow. *Essential of Behavioiural Research: Methods and Data Analysis*. McGraw-Hill Higher Education, New York, NY, USA, third edition, 2008.
26. A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.
27. W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Wadsworth Cengage Learning, 2002.
28. J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On breaking SAML: Be whoever you want to be. In *Proceedings of the 21st USENIX conference on Security symposium, Security*, volume 12, pages 21–21, 2012.
29. A. Stuchlik and S. Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM.
30. J. Stylos and B. A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.
31. J. Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, December 2013.
32. J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM.
33. D. F. Sutherland and W. L. Scherlis. Composable thread coloring. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 233–244, New York, NY, USA, 2010. ACM.
34. B. Ullmer and H. Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3.4):915–931, 2000.
35. B. J. Walther and A. S. Ross. The effect on behavior of being in a control group. *Basic and Applied Social Psychology*, 3(4):259–266, 1982.
36. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.
37. Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 60–68, New York, NY, USA, 2000. ACM.
38. H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.