Active Documentation: Helping Developers Follow Design Decisions

Sahar Mehrpour
Department of Computer Science
George Mason University
Fairfax, VA, USA
smehrpou@gmu.edu

Thomas D. LaToza
Department of Computer Science
George Mason University
Fairfax, VA, USA
tlatoza@gmu.edu

Rahul K. Kindi
Department of Computer Science
Cornell University
Ithaca, NY, USA
rkk59@cornell.edu

***Abstract*—Good documentation has long been argued to be key to helping developers write code more quickly and consistently with design decisions, but is left largely disconnected from code. We propose a method for active documentation, where design decisions are made explicit as design rules and checked against code. Developers can discover how to follow a design rule by navigating to examples in their codebase. After editing code, developers receive immediate feedback about which design rules are satisfied and which are violated, notifying developers who miss design decisions about the existence of these design decisions. We implemented our approach in a prototype tool and conducted a user study. Compared to developers using a traditional design document, developers working in an unfamiliar codebase with active documentation were faster and more successful, using active documentation to learn how to follow design decisions through examples and receive immediate feedback on their changes.**

***Index Terms*—documentation, design decisions, IDE**

I. Introduction

Software is the embodiment of the design decisions made by its developers, which together determine how software will satisfy its requirements [1]. A *design decision* captures a choice, constraining how software will be implemented, as well as *design rationale* explaining why this choice was selected. A *design rule* captures the specific choice made, expressing a constraint that must be true of an implementation to be consistent with the design decision [2]. For example, a developer might decide that, in order to implement a behavior, a specific class must have a reference to an object that it is contained within. Figure 1 illustrates how this design decision can be captured in two parts: a design rule expressing the constraint that a class (Microtasks) must have a reference to a containing object (Artifact) and text explaining the design rationale behind this choice.

Good documentation has long been argued to be the key to capturing design decisions and helping developers write code more quickly and consistently with the design [3]. Unfortunately, design documents today are largely exclusively a mix of text and figures that are entirely disconnected from code. To understand how to follow a design decision described in the document in practice, developers often seek to identify examples of the decision in code [4]. Lacking a direct connection from the documentation to code, developers must instead search through the code for examples, making it harder to learn how to follow decisions. Developers may fail to read the documentation at all or miss relevant design decisions and write code which violates the design. Unit tests might be used to catch some issues, but design decisions may not be behaviorally observable or may be at a level of abstraction that is not tested. Other issues may be caught during code review, but this process is not fullproof and offers no assistance to the developer as they are working in the task.

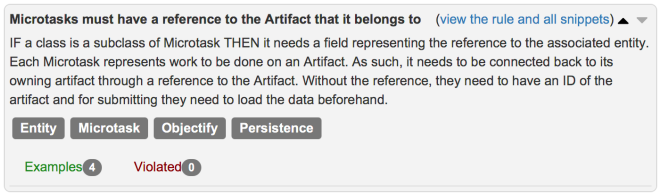

Fig. 1: In active documentation, a design decision is represented as a design rule, which can be checked against code, and a rationale, which explains the design choice.

Program analysis infrastructure offers rule checking engines which could be adapted to check design rules for conformance against code. *Extensible Static Checkers* such as CheckStyle [5], PMD [6], and FindBugs [7] identify potential defects in code by checking code for conformance against a set of rules. However, traditional rule checkers are specialized for addressing a different problem, offering the ability to check universal rules reflecting poor coding practice true of all projects rather than project-specific design decisions, focus on rules local to a file rather than rules where code in one file constrains code in another, and offer little support for explaining, organizing, and evolving rules used as documentation.

We propose a new form of documentation which is *active* in three senses. First, design rules expressing a design decision are translated into constraints and actively checked against code. As code or documentation changes, each is checked for conformance and divergences are flagged. Second, whenever a design rule applies to code, a link between the documentation and code is generated. Developers may then actively interact

978-1-7281-0810-0/19/$31.00

with the design rule, supporting the process of finding examples illustrating how to follow a design rule [8] by following links from documentation to code and supporting the process of understanding design decisions by identifying design rules and rationale related to the current code. Third, by connecting documentation to code, developers may more easily actively update and maintain documentation as both evolve.

In this paper, we argue that making documentation active enables developers to work more effectively with design decisions in code. We present a prototype, ACTIVEDOCUMENTATION, which embodies the active documentation approach and enables design decisions to be made explicit as design rules, checks design rules against code, and offers developers real-time feedback as they work (Section III). To evaluate our approach, we conducted a user study, where participants implemented a feature in an unfamiliar codebase (Section IV). The results suggest that active documentation enables developers to work more quickly and successfully by helping them understand where to start and how to follow design decisions. Our tool and study materials are publicly available.[1]

II. MOTIVATING EXAMPLE

In this section, we walk through an example of a developer interacting with active documentation in our system. Suppose Alice is working to build a crowdsourcing application, working on logic which generates microtasks. While adding a new feature, she finds herself in a portion of the codebase with which she is unfamiliar. Bob and other developers documented their design decisions as design rules, which were written with ACTIVEDOCUMENTATION and are available to Alice.

To find a place to begin, Alice first opens ACTIVEDOCUMENTATION in the IDE and reads the *Table of Contents* page displayed, skimming through several listed design rules. Looking to see if she might have already violated a rule in her implementation, Alice clicks on the *Violated Rules* tab (Figure 2). Seeing several violations, she clicks on the first violated rule. Three tags are associated with the rule, including 'sharding'. Not familiar with the concept, Alice clicks on the

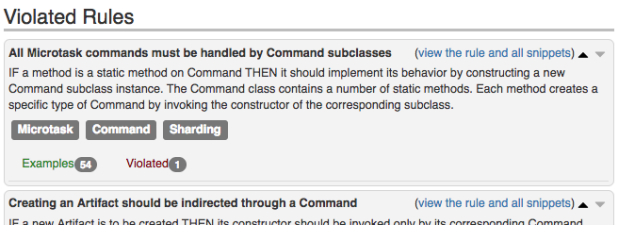

Fig. 2: The Violated Rules page lists rules with at least one violation.

gray 'sharding' tag, displaying the *Tag* page. (Figure 3). After reading the description, Alice understands that 'sharding' is about the system's data handling policy. Navigating back to the previous page, she reads the description of the violated rule once more, which now makes more sense. The rule states that all microtask actions (including the one she just created) must have their own subclasses, and these subclasses are grouped into 'command' classes, which she understands is necessary to ensure sharding works correctly. Clicking on the code snippet she sees listed as a violation (Figure 4), her editor navigates to the code she wrote which violates the rule.

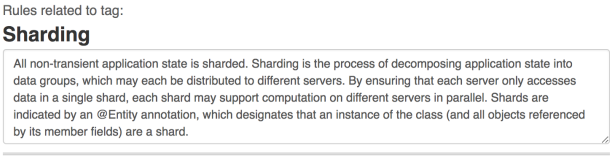

Fig. 3: Related design rules may be labeled with tags. Each tag is associated with an editable description, offering a place to explain concepts that crosscut individual design rules.

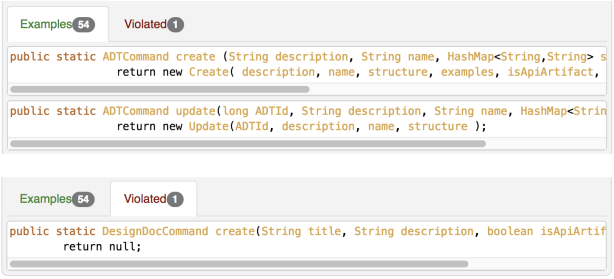

Fig. 4: Each design rule is accompanied by a list of example code snippets which satisfy (top) and violate the rule (bottom).

Alice notices that the line of code is returning `null`. After reading several code snippets listed as following the design rule in the *Examples* tab, she notes that all of these are generating an object by calling a constructor for a corresponding subclass. Following these examples, Alice adds the missing subclass, adds its constructor, and updates the return statement:

```
return new Create(title, descr, isApiArtifact, readOnly);
```

Alice saves her change. ACTIVEDOCUMENTATION updates, displaying a page listing all the rules applicable to her current file. One is listed with a green background; Alice notes that her change has caused the violated rule to now be satisfied (Figure 7). However, a second rule is indicated in red, indicating a new violated rule concerning a missing method. Alice adds the required method and saves her change. ACTIVEDOCUMENTATION again updates, indicating that the rule is now satisfied through a green background.

III. ACTIVE DOCUMENTATION

In this paper, we propose an approach for making documentation *active*, enabling design rules described in documentation to be directly checked against code for consistency, the ability to browse design rules connected to code or related to each other, the ability to find code examples illustrating how to follow documented design rules, and immediate continuous feedback notifying developers when they write code which

[1] bit.ly/ActiveDocumentation

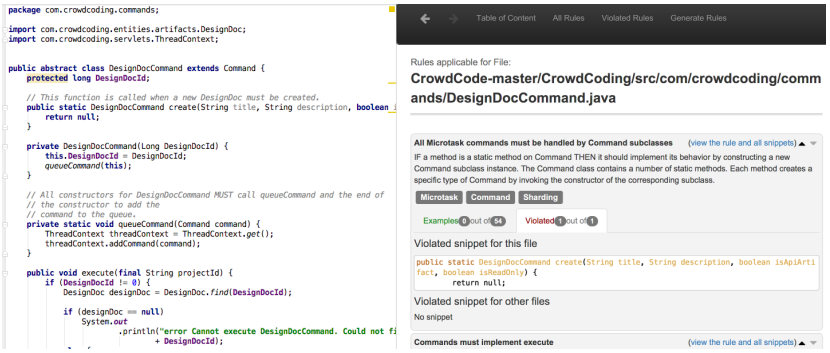

Fig. 5: ACTIVEDOCUMENTATION integrates design rules into the IDE, enabling rapid switching between code and related rules by displaying rules immediately adjacent to the code editor.

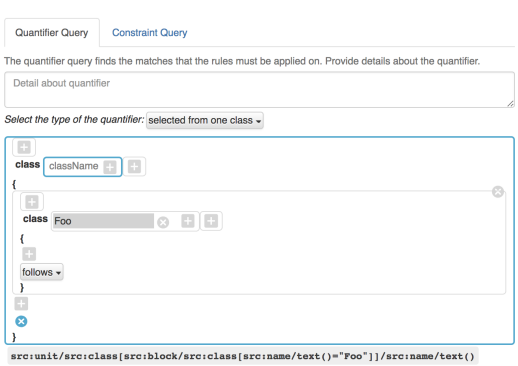

Fig. 6: The Generate Rules page enables developers to interact with a template of a design rule.

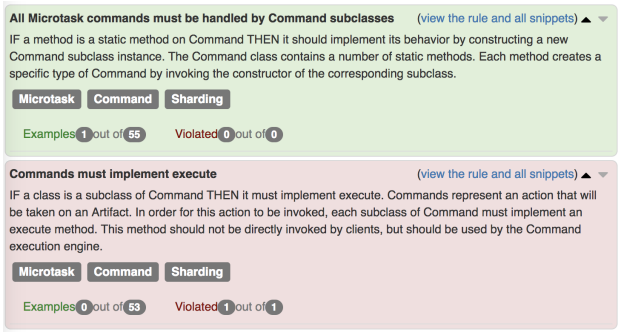

Fig. 7: ACTIVEDOCUMENTATION lists design rules which apply to the current file, indicating satisfied and violated rules with a green or red background, respectively.

violates a design rule. We implement our approach in a prototype tool called ACTIVEDOCUMENTATION.

To understand the types of design rules which might support developers work, we interviewed developers, asking them about the rules a developer should learn about code to successfully onboard. In addition, we examined two open source projects to extract design rules. From this, we built a small corpus of examples of design rules, which we then used to inform our choices about what types of design rules to support.

A. Interacting with design rules

ACTIVEDOCUMENTATION directly integrates design rules into the IDE, adding a design rules window to the development environment. This direct integration reduces the overhead of documenting design rules in a separate media [9]. Separate pages enable developers to browse all rules, see rules which apply to the active file in the code editor, see design rule violations across the whole codebase, and create new rules.

Design rules are displayed in panels, with a gray box grouping content related to a single design rule (Figure 1). To enable developers to understand the rationale behind the design rule, each rule includes an editable *title* and *description*. The description of the rule may include information about what the rule is, how it should be enforced, and the design rationale behind the rule. At the bottom of each rule panel are expandable tabs for viewing *example* snippets which satisfy the rule and *violated* snippets which violate the rule (Figure 4). Example snippets support developers in finding detailed examples of how to follow the rule [8]. Developers can click on a snippet to view the snippet and its surrounding code in the editor.

Design rules are often closely related, where several rules together implement a higher-level architectural or design decision. For example, a decision to shard data to enable parallelism might be implemented through a variety of design rules which describe how specific types of data are decomposed into individual groups. Related design rules are indicated through tags, enabling a design rule to be related to multiple higher-level decisions. Clicking on a tag anywhere in the interface opens a page listing all design decisions with the associated tag (Figure 3). The tag itself offers a description where developers can document the rationale behind the higher-level decision.

In large codebases, there may be many design rules, and developers need support for identifying rules relevant to the task at hand. When a developer switches the currently active file, a page is displayed which lists all of the rules which apply to code in the current file (Figure 5). To help developers find relevant information about the active file, the code snippets associated with the file are separated from the rest of snippets in the rule panels. Developers can use the All Rules page to browse all design rules in the project and the Table of Contents page to browse rules by tag.

As a developer edits code and design rules, the developer is offered continuous feedback on which rules are violated and which violations are fixed. If a change fixes a violation, the background of the corresponding rule panel turns green. If a new violation is created, the background of the panel turns red (Figure 7). Developers can browse rules with at least one violation in the Violated Rules page (Figure 2). In situations where developers import new code into a codebase or make larger changes, developers can see the impact of these changes in one place. In some cases, developers may choose to leave violations unfixed, introducing technical debt to implement

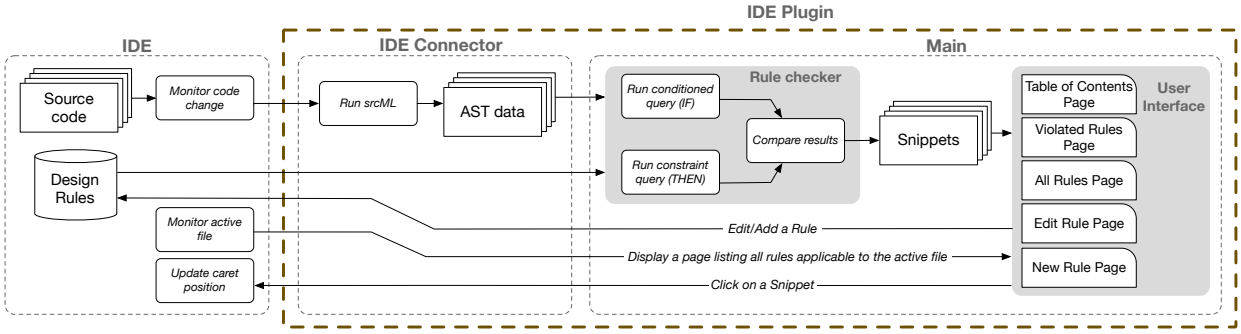

Fig. 8: The ACTIVEDOCUMENTATION system architecture. After a source code or design rule change, srcML generates an AST in an XML format. The rule checker then verifies each design rule against the AST by running two XPath queries, one for the quantifier and one for the constraint, generating a list of matches for each query. These matches are compared, generating a list of design rule examples and violations. These are then displayed to the user in several pages.

changes more quickly [10]. Developers working to reduce technical debt may then use the violated rules page to survey debt to be addressed.

Generating New Rules

Developers create design rules using the Generate Rules page. Each design rule is implemented as a set of abstract syntax tree (AST) queries. Developers can write AST queries using a visual interface, depicting a template describing which elements a query will match (Figure 6). The interface includes separate tabs for specifying the quantifier and the constraint queries. Developers are able to select elements participating in the rule from dropdowns in the template and set their properties through textboxes. The main elements and predicates of the query are distinguished by blue borders. For example to create a query for finding names of classes with a subclass named `Foo`, the developer can add a class in the body of the base class and set its name to `Foo` and mark the name of the base class as the target of the query (Figure 6). In this example, the design rule imposes constraints on a single class. Constraints may also be imposed on several classes. AST queries written in the visual interface are translated into XML Path Language (XPath) queries (displayed at the bottom of the page), and experienced developers may also write XPath queries directly, similar to PMD [6].

B. System architecture

ACTIVEDOCUMENTATION is implemented as a plugin for IntelliJ, a Java IDE [11]. The plugin is organized into two components: *IDE Connector* and *Main*. This design reduces dependencies on the underlying development environment, enabling a common Main component to be used with many environments and a small IDE connector to implement any environment-specific functionality. The IDE Connector is implemented in Java and interacts with the underlying environment to monitor code change and navigation events, retrieve source code, and navigate the code editor when required to go to examples. Most of ACTIVEDOCUMENTATION is implemented in JavaScript in the Main component, hosted as a web app running in a browser within the IDE.

In our approach, each design rule consists of two parts: a *quantifier* and a *constraint*. The quantifier specifies the scope in which the rule must be applied and the constraint specifies the condition that must be imposed. For example, consider the rule in Figure 1, *"IF a class is a subclass of Microtask, THEN it needs a field representing the reference to the associated entity"*. This rule imposes a condition on a specific set of classes. The quantifier and the constraint in this rule are determined by, respectively, the *IF* and *THEN* clauses in the description of the rule. In ACTIVEDOCUMENTATION, the rule quantifier and the rule constraint are each translated into AST queries. Compared to PMD [6] which is restricted to a single XPath query per rule, ACTIVEDOCUMENTATION is able to check more complex rules that require multiple sub-queries. In these cases, the result of a sub-query is used as input into a second sub-query, which may be further chained. As sub-queries are chained with element identifiers rather than fully qualified names, there may be false positives in chained sub-queries when multiple elements share the same identifier.

Design rules are checked against code in several steps (Figure 8). The IDE Connector first receives the source code from the IDE. Next, it uses srcML to generate an abstract syntax tree of the source in XML format [12]. The srcML formatted data and the design rules are then sent to the Main component for processing. The Main component executes XPath queries defined for each design rule on the designated files and folders. Each query generates a set of elements which match the quantifier and the constraint. The Main component then compares these sets to generate a set of examples and violations for each rule. The Main component generates code snippets from this result and updates the current page with the new data, if applicable.

The Main component generates requests for the IDE Connector based on user interactions. Some of these requests are then passed to the IDE. When a code change or navigation event occurs, the IDE Connector notifies the Main component. For example, when a user clicks on a code snippet, the Main component sends a request to the IDE Connector to navigate the code editor. When a developer defines a new rule, the Main

component sends all information on the new rule to the IDE Connector, which persists the data as a JSON file. Figure 8 illustrates the ACTIVEDOCUMENTATION system architecture.

IV. EVALUATION

A key role of documentation is in helping developers become familiar with a codebase. As developers ultimately put this knowledge to use in implementing features, we conducted a lab study in which we asked developers to implement a small feature. In our evaluation, we sought to understand the impact of making documentation active by conducting a between-subjects study comparing developers working with traditional documentation with those using active documentation.

A. Method

We recruited 21 participants from graduate students enrolled in a software engineering graduate course with prior experience in Java. Participants ranged in programming experience from 2 to 20 years (median 5 years). Most had experience as a professional software developer, ranging from 0 to 15 years (median 2 years). All were familiar with at least one IDE. As the task involved writing code interacting with a persistence framework, we asked participants about their experience with persistence frameworks. 8 participants reported familiarity with a persistence framework (1 expert, 1 moderately familiar, 3 somewhat familiar, and 3 slightly familiar).

At the end of the study, we interviewed participants about their documentation practices. Participants varied in the reported frequency with which they document their project, although almost all reported they add comments in code, particularly code they found potentially complicated and confusing. Participants reported using JavaDoc, using tool support to generate templates for function declarations. Some participants reported authoring documentation using external applications such as version control tools and text editors.

We adapted a task from the implementation of CrowdCode, a web-based IDE for developing projects using *microtasks* [13] (described briefly in the Motivating Example). Participants were expected to interact with only the back-end of the system, which includes approximately 9000 lines of Java in 107 classes. A key abstraction is an *artifact*. Artifacts are persisted in the Google App Engine persistence framework [14]. Participants were given a partial implementation of a new artifact and were asked to continue its implementation by focusing on storing it to the persistence store. As persistence was a crosscutting concern, this required adding approximately 20 lines of code and modifying 2 lines of code in 4 different classes. All participants were given access to the same documentation, describing design rules about how artifacts were implemented and how they were persisted to the data store. We extracted 15 design rules from the codebase, which we gave to all participants.

Participants were randomly assigned to one of two conditions, a control group (P1-P11) and an experimental group (P12-P21). Participants in both conditions had a similar distribution of programming experience (median 6 years for the control group and 5 years for the experimental group) and professional experience (median 2 years for both groups). Experimental participants were given access to ACTIVEDOCUMENTATION, containing the design rules for the project. Control participants were given a traditional design document listing the same design rules in text including titles and descriptions of the rules as well as the concepts each was related to.

After beginning the study, participants in both groups first completed a warmup task to familiarize themselves with the IDE. Experimental participants familiarized themselves with ACTIVEDOCUMENTATION as well.

All participants were then given 70 minutes to complete the main task. As participants worked, we asked participants to think aloud and captured a screen and audio recording. At the end of the study, we collected their final code and conducted a semi-structured interview. We asked participants about how often they document their code, methods they use to document, and prior experience with tools offering real-time feedback. We asked participants in the control condition about challenges they had when using the design document, while we asked participants in the experimental condition about their perceived usefulness of ACTIVEDOCUMENTATION, challenges, and desired features. We removed the data of three participants (two from the control group and one from the experimental group) who found the task too difficult and gave up early in the task, resulting in a total of 9 participants in each condition.

B. Results

To investigate the impact of ACTIVEDOCUMENTATION on task time, we measured the task time for each participant from when they first had access to the code to the point where they announced the completion of the task or reached the maximum allotted time. Overall, participants with ACTIVEDOCUMENTATION were significantly faster (Mann Whitney U test, $U = 16.5$, $p < 0.05$), finishing the task in an average of 49 minutes (*Task Duration* column in Table I). Participants in the control condition completed the task in an average of 68 minutes. 7 of the 9 participants in the control condition used the entire task time, resulting in a much higher variation in task times for participants with ACTIVEDOCUMENTATION ($SD = 17.4$) than for control participants ($SD = 3.4$).

To assess participants' success in the task, we examined the code written by participants. As described in Section IV-A, the task required participants to implement code to persist a new artifact class. To evaluate participants' implementations, we ran the final implementation created by each participant, inspected the persisted data, and scored the work as successful if it correctly persisted all of the data. We found that participants with ACTIVEDOCUMENTATION were significantly more successful (Fisher's exact test, $p < 0.027$). Only 23% of control participants (2 of 9) succeeded while 78% of participants (7 of 9) with ACTIVEDOCUMENTATION succeeded (indicated by green rows in Table I). To quantify how close participants who did not succeed were to finishing, we edited each submission to complete the implementation, modifying any lines as necessary to fix defects. Participants in both

	Diff		Time (Minutes)		Submitted Lines of Code		
	Added	Removed	First Edit	Task Durat.	Missing	Incorrect	Task Irrlvnt
Control Group							
P1	15	5	14	70	16	1	8
P2	65	2	16	70	0	3	18
P3	124	12	4	70	16	1	87
P4	2	1	64	70	17	2	1
P5	36	3	11	61	0	0	7
P6	23	2	8	64	1	0	0
P7	5	1	43	70	16	1	2
P8	69	3	11	70	0	0	34
P11	66	3	12	70	1	8	29
Mean	45.00	3.56	20.33	68.33	7.44	1.78	20.67
Median	36.00	3.00	12.00	70.00	1.00	1.00	8.00
Std. Dev.	39.64	3.40	19.82	3.39	8.37	2.54	27.76
Experimental Group							
P13	21	19	13	70	16	1	29
P14	29	3	8	49	0	0	0
P15	29	4	2	70	0	0	4
P16	33	2	6	39	1	0	2
P17	26	2	1	27	0	0	0
P18	44	3	6	70	0	0	15
P19	28	3	9	40	0	0	0
P20	31	2	4	28	0	0	2
P21	26	2	8	47	0	0	1
Mean	29.67	4.44	6.33	48.89	1.89	0.11	5.89
Median	29.00	3.00	6.00	47.00	0.00	0.00	2.00
Std. Dev.	6.36	5.50	3.71	17.44	5.30	0.33	9.87
All Participants							
Mean	37.33	4.00	13.33	58.61	4.67	0.94	13.28
Median	29.00	3.00	8.50	70.00	0.00	0.00	3.00
Std. Dev.	28.65	4.46	15.59	15.77	7.37	1.95	21.59
p or U value	0.142	0.343	**12.5**	**16.5**	0.056	**0.043**	0.082

TABLE I: The lines of code which participants submitted which were missing, incorrect, or modified but task-irrelevant; the total added and removed lines of code; and the task duration and time of first edit. The last row indicates U values for the First Edit and Task Duration (with $p < 0.05$) and p values for the remaining columns (Significant values are bold). Green rows indicate participants who finished the task successfully.

groups who were not successful were bimodal in the amount of work which remained: several needed only a single line while many needed 16 or 17 additional lines (shown in the Missing and Incorrect columns in Table I)

One potential benefit documentation may offer is in helping developers to more quickly and successfully understand what code is necessary to implement the desired feature. That is, effective documentation might be expected to reduce the amount of task-irrelevant code developers write and reduce the amount of necessary code that developers miss and fail to write correctly. To analyze this, we first created a diff of each participant's change and calculated the number of added and removed lines (shown in the *Diff* columns in Table I). There was no significant difference between conditions. Next, we labeled each changed line as being either *task-related* or *task-irrelevant* by comparing the change against a correct change and identifying additional lines unrelated to the feature (shown in the *Submitted Lines of Code* columns in Table I). For task-related lines, we compared the change against the correct change and identified lines that were incorrect. Finally, to understand if participants varied in how fast they were able to get started with their implementation, we recorded and analyzed the time at which participants first edited the code.

Participants with ACTIVEDOCUMENTATION added significantly fewer lines of incorrect task-related code (one-tailed unequal t-test, $p < 0.044$). On average, code written by participants with ACTIVEDOCUMENTATION was missing fewer task-relevant lines of code, but the difference was not significant (one-tailed equal t-test, $p < 0.057$). Participants with ACTIVEDOCUMENTATION modified fewer lines of task-irrelevant code, changing on average 5.89 lines of code compared to 20.67 lines of code for control participants. However, the difference was not significant, in part due to two outliers (P13 and P18) which we discuss below.

Participants with ACTIVEDOCUMENTATION (Median=6) were able to start editing code significantly faster than the control participants (Median=12) (Mann Whitney U test, $U = 12.5$, $p < 0.05$). On average, participants with ACTIVEDOCUMENTATION made their first edit after only 13.33 minutes, while participants in the control group made their first edit after 20.33 minutes. The *First Edit* column in Table I lists the time at which each participant made their first edit.

To help understand these results and identify ways in which ACTIVEDOCUMENTATION may have enabled developers to work differently, we analyzed the screen recordings and think aloud data as well as the post-task interview data, identifying differences in behaviors we observed and challenges that participants reported experiencing. Control participants reported experiencing challenges finding relevant design decisions within the design document and connecting code with these design decisions. For example, P1 and P5 reported that they believed that the document was unnecessarily long and thus hard to work with, and P2 reported that he had difficulty understanding the relationships among rules. In contrast, participants with ACTIVEDOCUMENTATION reported viewing the *Violated Rules* page to find relevant design decisions.

> "(Looking at Violated Rules page) Oh nice ... this is kind of giving me how to solve [the task]." - P13

As control participants continued work, the task required them to work with code scattered across several classes. While some locations were easy to identify, others required more effort, and many struggled. Participants with ACTIVEDOCUMENTATION used the violated snippets to identify relevant places to make changes. They also used example snippets listed for each design rule in ACTIVEDOCUMENTATION to compare examples of the rule and the faulty lines of code.

> "(clicking on an *Example* code snippet, the caret moved to the location of the snippet in the code) ... That is nice ... Seeing that something is working correctly ... is very helpful." - P12

Real-time feedback helped ACTIVEDOCUMENTATION participants to detect errors and violations early. Participants reported

that the real-time feedback was valuable, offering feedback about rule verification immediately after changing the code without running the application.

To explore why some participants seemed to benefit less than others from ACTIVEDOCUMENTATION, we investigated the submitted code and screen recordings by the two participants with the highest number of modified lines in the experimental group. We observed that P13 often failed to use the features offered by ACTIVEDOCUMENTATION. Instead of using the feature linking snippets to code, the participant visually searched for snippets illustrating violations and examples in the code, and as a result, the participant made changes in a wrong class. In contrast, P18 was overeager in their work, identifying a violation of a design rule in the code that was unrelated to the task. After finishing the task, they continued by working to implement a fix to this issue.

The study revealed several usability issues with ACTIVEDOCUMENTATION. While the tool was loading, many participants refreshed the page, which restarted the tool and caused the tool to be non-responsive while it finished loading. Some participants appreciated that the listed rules in ACTIVEDOCUMENTATION constantly changed as they navigated through code and used this to verify each file against design rules. Others were frustrated by the constant view changes and wished to have more direct control. Participants suggested a number of additional features, including flagging rules for revision (P12), displaying file names for each snippet (P13), adding markers in the code editor to indicate the presence of satisfied and violated design rules (P17), adding search based on keywords in the code and documentation (P15), and automatic code fixes (P16). Some participants suggested that enabling design rules to be hidden or shown on demand may help them to spend their time on rules relevant to the task at-hand.

V. LIMITATIONS AND THREATS TO VALIDITY

As in all studies, our study has several important limitations and threats to validity. Participants worked on a codebase with which they were unfamiliar, as in all programming lab studies. Much of the task involved working with a specific persistence framework, with which developers were unfamiliar. Developers with more experience in a codebase or a framework might benefit less, as they might already have internalized more of the design rules. In practice, this may occur when developers are new to a team or find themselves working in part of a large codebase with which they are previously unfamiliar.

Participants were artificially limited in learning about the codebase exclusively by reading the code and documentation. In practice, developers may have access to other resources, such as asking experienced teammates or the original authors to go over the code with them and explain aspects of the design. However, developers often do rely first on building their own understanding, as interactions with others can impose costs on others and developers are often expected to have first made an effort to understand themselves [15].

Finally, our study focused on the benefits developers might gain from active documentation that is already created. A key problem our study did not examine is in helping developers create and modify documentation to maintain its consistency with the code. This is an important subject for future work.

VI. RELATED WORK

Documenting design decisions has long been viewed as important to maintaining the comprehensibility of code [16]. Keeping the documentation in sync with source code is one of the key challenges developers face during software development [17]. Developers often fail to update the documentation regularly and completely, resulting in outdated, incomplete, and untrustworthy documentation [18]. Poorly-written and missing documentation is a significant cause of defects [19] and a serious challenge newcomers face [20]. Developers often consider API documentation errors and incompleteness as blocking issues which force them to use other APIs [21]. Many tools have explored approaches for documenting various aspects of software. For example, there are tools for capturing architecture decisions [22], decision knowledge [23], and implementation decisions [24]. Tutorons generates context-relevant on-demand micro-explanation of code [25]. Design fragments document repeated patterns in code about how a program interacts with a framework [26]. Robillard et al. also studied on-demand developer documentation [27].

Throughout their interactions with code, developers interact with crosscutting concerns, reflecting decisions whose implementation is scattered through code. Working with crosscutting concerns in traditional IDEs is problematic: simply navigating dependencies through code may occupy a third of developers' time [28] and developers can become disoriented and lost [29]. In response, a wide range of languages and systems have explored helping developers to more effectively work with scattered code, reifying concerns into various constructs. Language-based solutions such as aspect-oriented programming reify scattered code as a new entity, an aspect, that is weaved into the source at compile-time [30]. Concern browsers (e.g., [31]) collect lists of artifacts related to a concern, supporting navigation between artifacts. Other systems change how developers interact with the IDE itself, displaying concerns as snippets of code in "bubbles" displayed in diagrams [32]. Tools such as aspect miners support queries for locating scattered code [33]. Active documentation builds on these ideas by offering similar navigational benefits, reifying concerns into groups of related design rules and supporting navigation between code snippets using rules. It extends these ideas by associating a navigational context with an explicit design rule and checking these rules against code.

Developers document rationales for decisions that are obscure or associated with other decisions [34]–[36] and they often ask about the rationale behind design decisions [37]. A number of sophisticated tools exist for documenting rationales within the IDE [38], building on research exploring the structure and representation of rationales. Design rationale systems largely focus on documenting alternatives and reasons for choices between alternatives rather than checking if code is

conformant with a decision. In active code completion, developers are able to design 'palettes' for defining rules, access them through auto complete menus, and insert code based on the rules [39]. In Codelets, developers are able to search for example code previously uploaded in the tool, and edit and insert them in code [40]. Unlike ACTIVEDOCUMENTATION, such systems do not actively check new code written by the developer for conformance to a design rule.

Other work has investigated documenting constraints imposed by decisions and checking that code conforms to these constraints. Broadly, lightweight specification and verification can often be viewed as checking that decisions imposed by an API are satisfied by an API client. Much of this work captures constraints imposed in an interface or contract, and then ensures client code conforms to this contract [41]. Other work considers constraints to be crosscutting, checking design rules [42] or inspecting code for incorrect idioms [7]. However, such constraints reflect constraints imposed by an API, not those made by client developers, and often focus on design decisions that are behaviorally observable. Another approach is to generate constraints from higher-level artifacts, using them to check that code correctly conforms to an architecture [43] or design model [44], [45]. Caracciolo at al. studied conformance checking of architectural rules specified by *Dicto*, a domain specific language [46]. Existing work has not explored how design decisions more generally might be translated into constraints and checked in code.

Extensible rule checkers support finding potential defects in code at the statement level. Coding styles can be checked using CheckStyle [5]. All of these tools are intended to check for common defect patterns true of all projects rather than as a method for documenting project-specific design decisions. Other tools focus on analyzing the architecture of code [47], such as by using dependency-structure matrices [48], source code query languages [49], and reflexion models [50].

VII. DISCUSSION

Maintaining documentation consistently with code is challenging. In traditional documentation, developers do not receive any notification when documentation and become inconsistent. In this paper, we proposed an approach for making documentation active. In active documentation, documentation is checked against code for conformance, a link between design rules and code is generated, and developers can view divergences between code and design rules and update either. We explored this approach by implementing a tool which directly integrates design rules with code, offering developers examples of each rule and immediate feedback when rules are violated. We found that, compared to a traditional design document, developers offered the same text as active documentation were able to work significantly more quickly and successfully. Developers used active documentation to get started implementing code faster, learn how to follow design decisions by finding examples, and receive immediate feedback on changes before running the code.

As we observed in our study, developers working with a design rule often seek to identify example code that can be used to satisfy the design rule and understand what parts of the example are required. ACTIVEDOCUMENTATION helped support developers in this work, offering a list of short code snippets and enabling developers to navigate the code editor to the full example by clicking on the code snippet. However, even with this support, developers still faced challenges working with examples. In particular, it was sometimes challenging for developers to extract from the code the minimal elements necessary to satisfy the design rule. Better editor support might help address this problem by directly highlighting the elements in the code which match the quantifier and constraint queries. Alternatively, the code snippets view might display an abstract example with just the query constraints, which developers could copy and paste into code. Yet, in some cases these examples might be too minimal, as developers may sometimes wish to see typical accompanying code. Techniques for finding patterns and generalizing examples may offer solutions [51].

Design rules come in many forms. In our prototype, we chose to represent design rules as patterns in the AST, enabling the system to check several types of design decisions. The concept of active documentation as well as the system itself are agnostic to the underlying rule checker and require only an engine which can take a set of rules and output a set of code examples which satisfy and violate the rules. Other forms of static analysis based rule checkers could be directly integrated into our approach. For example, a protocol checker might directly check that an initialization method is called before other methods. Mechanisms would also be needed to enable developers to author design rules quickly and easily, likely through interactive interfaces rather than by directly writing new analyses in a static analysis framework. Another approach to checking rules may be to use design rules to generate corresponding tests. However, waiting for tests to execute could introduce delays in obtaining feedback, requiring ways to signal that some design rules have not yet been verified.

A key challenge for all documentation systems is incentivizing developers to document their design decisions. In traditional documentation, the developer writing the documentation may receive no immediate benefit from the long-term investment of writing the documentation. By making design rules active, the environment may offer some immediate benefits. Writing the design decision down as a design rule enables the developer to check if it is satisfied by the existing code, helping the user decide if the decision should be abandoned or to update violations to conform with the decision. Writing a design rule also helps the developer find examples, potentially finding more details on how code elsewhere follows the design constraint. Our prototype offers a basic editor for authoring design rules. The usability of the authoring experience is clearly key to reducing the cost incurred by a developer when they write down a design decision. More work remains to improve and simplify the process of authoring design rules.

ACKNOWLEDGMENTS

We thank our study participants for their time. This work was supported in part by the National Science Foundation under grant NSF CCF-1703734.

REFERENCES

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity*. MIT Press, 2000, vol. 1.

[3] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, 1986.

[4] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681.

[5] "CheckStyle," http://checkstyle.sourceforge.net, 2004.

[6] T. Copeland, *PMD Applied*. Centennial Books, 2005.

[7] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *ACM SIGPLAN Notices*, vol. 39, no. 12, 2004, pp. 92–106.

[8] A. Head, C. Sadowski, E. Murphy-Hill, and A. Knight, "When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects," in *International Conference on Software Engineering (ICSE)*, 2018, pp. 643–653.

[9] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[10] W. Cunningham, "The WyCash Portfolio Management System," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1992, pp. 29–30.

[11] JetBrains, "IntelliJ IDEA," https://www.jetbrains.com/idea/, 2000–2018.

[12] J. I. Maletic, M. L. Collard, and A. Marcus, "Source code files as structured documents," in *International Workshop on Program Comprehension*, 2002, pp. 289–292.

[13] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. Van Der Hoek, "Microtask programming: Building software with a crowd," in *Symposium on User Interface Software and Technology (UIST)*, 2014, pp. 43–54.

[14] "Google App Engine," https://cloud.google.com/appengine/, accessed: 2018-07-10.

[15] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *International Conference on Software Engineering (ICSE)*, 2006, pp. 492–501.

[16] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[17] R. Pierce and S. Tilley, "Automatically connecting documentation to code with rose," in *International Conference on Computer Documentation*, 2002, pp. 157–163.

[18] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Software*, no. 6, pp. 35–39, 2003.

[19] M. Visconti and C. Cook, "Software system documentation process maturity model," in *Conference on Computer Science*, 1993, pp. 352–357.

[20] I. Steinmacher, C. Treude, and M. Gerosa, "Let me in: Guidelines for the Successful Onboarding of Newcomers to Open Source Projects," *IEEE Software*, 2018.

[21] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[22] C. Manteuffel, D. Tofan, P. Avgeriou, H. Koziolek, and T. Goldschmidt, "Decision architect–A decision documentation tool for industry," *Journal of Systems and Software*, vol. 112, pp. 181–198, 2016.

[23] T.-M. Hesse, A. Kuehlwein, and T. Roehm, "DecDoc: A tool for documenting design decisions collaboratively and incrementally," in *International Workshop on Decision Making in Software ARCHitecture*, 2016, pp. 30–37.

[24] T.-M. Hesse, A. Kuehlwein, B. Paech, T. Roehm, and B. Bruegge, "Documenting Implementation Decisions with Code Annotations," in *International Conference on Software Engineering and Knowledge Engineering*, 2015, pp. 152–157.

[25] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 3–12.

[26] G. Fairbanks, D. Garlan, and W. Scherlis, "Design Fragments Make Using Frameworks Easier," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006, pp. 762–763.

[27] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez *et al.*, "On-demand developer documentation," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 479–483.

[28] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," in *International Conference on Software Engineering (ICSE)*, 2005, pp. 126–135.

[29] B. De Alwis and G. C. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2006, pp. 51–54.

[30] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2001, p. 313.

[31] M. Kersten, M. Chapman, A. Clement, and A. Colyer, "Lessons learned building tool support for AspectJ," in *International Conference on Aspect-Oriented Software Development (AOSD) Industry Track*, 2006.

[32] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, "Code bubbles: a working set-based interface for code understanding and maintenance," in *Conference on Human Factors in Computing Systems (CHI)*, 2010, pp. 2503–2512.

[33] A. Kellens, K. Mens, and P. Tonella, "A survey of automated code-level aspect mining techniques," in *Transactions on Aspect-Oriented Software Development IV*, 2007, pp. 143–162.

[34] A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech, "Rationale management in software engineering: Concepts and techniques," in *Rationale Management in Software Engineering*, 2006, pp. 1–48.

[35] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, and P. Kruchten, "The value of design rationale information," *Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 21:1–21:32, 2013.

[36] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. A. Babar, "A comparative study of architecture knowledge management tools," *Journal of Systems and Software*, vol. 83, no. 3, pp. 352–370, 2010.

[37] T. D. LaToza and B. A. Myers, "Hard-to-answer Questions About Code," in *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010, pp. 1–6.

[38] J. E. Burge and J. D. Kiper, "Capturing decisions and rationale from collaborative design," in *International Conference on Design Computing and Cognition*, 2008, pp. 221–239.

[39] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *International Conference on Software Engineering (ICSE)*, 2012, pp. 859–869.

[40] S. Oney and J. Brandt, "Codelets: Linking interactive documentation and example code in the editor," in *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2012, pp. 2697–2706.

[41] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[42] C. Morgan, K. De Volder, and E. Wohlstadter, "A static aspect language for checking design rules," in *International Conference on Aspect-Oriented Software Development (AOSD)*, 2007, pp. 63–72.

[43] R. S. Silva Filho, F. Bronsard, and W. M. Hasling, "Experiences documenting and preserving software constraints using aspects," in *International Conference on Aspect-Oriented Software Development (AOSD)*, 2011, pp. 7–18.

[44] S. P. Reiss, "Constraining software evolution," in *International Conference on Software Maintenance*, 2002, pp. 162–171.

[45] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, no. 2, pp. 188–204, 2010.

[46] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "A unified approach to architecture conformance checking," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2015, pp. 41–50.

[47] L. Passos, R. Terra, M. Tulio Valente, R. Diniz, and N. Mendoça, "Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.

[48] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 167–176.

[49] O. De Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote Address: .QL for Source Code Analysis," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2007, pp. 3–16.

[50] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007, pp. 12–12.

[51] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing API Usage Examples at Scale," in *Conference on Human Factors in Computing Systems (CHI)*, 2018, pp. 580:1–580:12.