If this paper is accepted, please only make it available to workshop attendees.

UNAKITE: Support Developers for Capturing and Persisting Design Rationales When Solving Problems Using Web Resources

Michael Xieyang Liu*, Nathan Hahn*, Angelina Zhou†, Shaun Burley‡, Emily Deng†, Jane Hsieh§, Brad A. Myers*, Aniket Kittur*
*†‡Human-Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA, USA
§Oberlin College, Oberlin, OH, USA
{xieyangl, nhahn, bam, nkittur}@cs.cmu.edu*, {ajzhou, edeng}@andrew.cmu.edu†, me@shaunburley.com‡, jhsieh@oberlin.edu§

***Abstract*—UNAKITE is a new system that supports developers in collecting, organizing, consuming, and persisting design rationales while solving problems using web resources. Understanding design rationale has widely been recognized as significant for the success of a software engineering project. However, it is currently both time and labor intensive for little immediate payoff for a developer to generate and embed a useful design rationale in their code. Under this cost structure, there is very little effective tool support to help developers keep track of design rationales. UNAKITE addresses this challenge for some design decisions by changing the cost structure: developers are incentivized to make decisions using UNAKITE's collecting and organizing mechanisms as it makes tracking and deciding between alternatives easier than before; the structure thus generated is automatically embedded in the code as the design rationale when the developer copies sample code into their existing code. In a preliminary usability study developers found UNAKITE to be usable for capturing design rationales and effective for interpreting the rationale of others.**

I. INTRODUCTION

Programming is a high cognitive load activity [1]–[4], where developers continually make hypotheses, propose questions, and discover answers. Developers must understand existing code written by others, determine how to write new code based on a large number of constraints and requirements, or select among a set of application programming interfaces (APIs) and software development kits (SDKs).

All of these cognitive tasks can be classified as attempts by the developers to gain knowledge about their code, APIs, options, requirements, etc. This knowledge is then often used to make a decision to solve a problem. The reason behind the decision, called the "design rationale" can be important later to understand *why* the decision was made [5]–[7]. In general, design rationale documents the decisions made during a design or investigation process (i.e., "whether to use React.js or Angular.js to build a website"), and the considerations behind those decisions, (i.e., "what are the alternatives that might be used?", "what constraints and requirements should be met?", "which option was chosen and why?", etc.). When developers are trying to later understand and change existing code, the answers to these questions can be crucially important, for example so the new developer does not accidently violate important constraints that guided the original developer. Indeed, questions about design rationale are among the most frequently asked by developers [1], [8], and are, unfortunately, also the ones that lack effective tools to directly support.

(a) *special comment* that contains an auto-generated unique id (left) & the corresponding green block in the overview ruler (right); (b) *popup window* that contains the essential **design rationale** embedded in the project. The window reveals itself automatically when the cursor hovers over the special comment.

Fig. 1: UNAKITE in action in Visual Studio Code editor

Significant prior research focuses on systems and tools to help developers *find* their needed initial knowledge by augmenting existing general-purpose search engines with programming specific information filters [9], [10], mining software repositories and online forums to form example-centric code searches [11]–[13], or trying to offload the search process to a remote helper [14]. However, we argue that finding the right knowledge is just the first step: the developer must still *organize* that knowledge to make a decision to solve their problems (e.g., looking at the discovered features of the APIs to decide which one to use). Furthermore, after each

sensemaking episode in which developers gain knowledge and make a decision, their work is typically lost due to the absence of an easy way to keep track of it. Tool support is needed to help developers to digest the knowledge learned in order to make an educated decision, as well as to provide mechanisms to keep track of the associated design rationale for future reference.

To guide the design of our tool, we performed an initial qualitative study with developers about their current approaches to discovering and tracking design information when using web resources. Using these qualitative findings together with issues and suggestions indicated by prior work, we designed and implemented UNAKITE (see Fig. 1), which stands for "**U**sers **N**eed **A**ccelerators for **K**nowledge for **I**mplementations in **T**echnology **E**nvironments" (unakite is also a green and pink semi-precious stone).

The core challenge we address in UNAKITE is the cost structure of generating and tracking design rationale for certain decisions. Currently, it is time and effort intensive for little immediate payoff for a developer to embed a useful design rationale in their code. Not only is payoff in the future, but is largely for the benefit of others, with uncertainty about whether it will be valuable, relevant, or even comprehensible. UNAKITE addresses this challenge for decisions about what to use by changing the cost structure of the task: as developers use the web to search for information, they are incentivized to make decisions using UNAKITE's collecting and organizing mechanisms as it makes tracking and deciding between alternatives easier than having to hold them in the users' working memory. By externalizing their thought processes [15], UNAKITE helps ease developers' cognitive effort of transforming raw information into decisions and design rationales, and eliminates the effort that would be required for writing documentation after decisions are made.

Next, the structure thus created with UNAKITE is saved for the future and automatically attached to the code if developer copies from code examples into their existing code, thus supporting *persisting* this knowledge. Later developers unfamiliar with the code upon encountering the relevant parts can see what alternatives were considered, what dimensions or constraints they were compared on, and what evidence and web resources the original developers used in their decisions.

To evaluate whether all of this might work, we performed a preliminary usability study that showed that UNAKITE is usable, effective, and desirable among the participants.

The contributions described in this paper are:

- Results of a formative study about how developers capture and use web-based information to make decisions.
- A fluent way of *collecting* programming-specific knowledge on the Internet and categorizing that knowledge into options, requirements, and information snippets pertaining to the investigation task.
- A novel way of *organizing* the knowledge to support decision-making along with the design rationales for those decisions, which offers both the ability to automatically generate a comparison table of the various options, and to manually customize and refine the information according to the users' needs.
- A new way of embedding design rationales for these kinds of decisions, in addition to the source URLs, into existing code projects that minimizes the effort to make sense of the original decisions.
- UNAKITE, a system that integrates the above features together, along with a user study suggesting their value for capturing and understanding design rationale.

II. Related Work

The concept of "design rationale" dates back at least to Kunz and Rittel's [16] use in 1970 for supporting planning and coordination of political decision processes. A 1992 paper surveys design rationales from a software engineering perspective and discusses different models used in design rationale systems [5]. Arora et al. later introduced a framework for capturing design rationale using granularity hierarchies [6]. These representational models established the significance of capturing design rationales for the success of a software engineering project, but they do not provide tools to create and persist the rationale, like UNAKITE.

More recent research has identified that understanding the intent and rationale for why code was done in a particular way is one of the hard-to-answer questions for developers [1], [8], [17]. LaToza et al. suggested that developers often try to understand the design rationale for surprising decisions by trying to deduce the possible motivating requirements and criteria [18]. Ko et al. reported that developers often speculate about the correctness and legitimacy of a decision, and they often wish to see the alternatives related to a decision [4].

Despite the prevalence of the problem, effective support for understanding design rationale is still considered an open question. LaToza et al. suggested that while some of the questions could be tackled by changing and testing the code itself, the majority of design rationale questions are difficult to answer in this fashion due to their non-functional nature [1]. Asking colleagues and team mates might help ease the underlying concern about design decisions, alternatives, and requirements that are nearly impossible to test [17], but often the original designer is not available, and even when they are, it causes interruptions to the workflow of both the help requester and giver. That research also shows that knowing the original investigation task behind a design decision is a critical step towards understanding design rationale, motivating UNAKITE to provide that information.

Comments serve the purpose of improving source code readability, and are considered valuable for code understanding and maintenance [19], [20], suggesting it being a good place to capture design rationales [1]. However, it is well-known that developers do not like to write comments even if they are simple [21], and do not trust comments to be up-to-date [22]. Further, the author of the design rationale has to anticipate what concerns the readers might have and provide relevant answers. Another recent trend for design rationale is through the form of storytelling. For example, Storyteller is a tool that

permits developers to annotate software projects with rich media like pictures and audio [23]. Attaching the design rationale in code through commenting and storytelling maximizes the convenience and benefits for later developers to locate and comprehend them. But, the fundamental challenge remains that it is both time and effort intensive for the initial developers to contribute a useful piece of design rationale. UNAKITE addresses this challenge at least for a certain class of decisions, by leveraging the initial developers' decision making process. UNAKITE does not currently have any approach for addressing the issue of comments going stale, but even if they are out-of-date, we feel it may be useful for developers to at least know the original design rationale.

III. FORMATIVE INVESTIGATION

We conducted a qualitative interview study with developers to better understand their current approaches for discovering and tracking design rationale while solving problems in real programming projects. We interviewed 15 developers (11 male, 4 female) recruited from the authors' social networks. In order to capture a variety of processes, we chose 5 professional software developers, 2 doctoral students and 8 master students. While we do not claim that this sample is representative of all developers, the interviews were very informative and helped motivate the design of our system.

A. Interview

The interviews lasted up to 30 minutes and were semi-structured. Participants were not compensated for their time. After basic demographics questions, the participants were asked to retrospectively describe at least one recent difficult coding task they attempted to solve that involved heavy searching and sensemaking on the Internet. To facilitate accurate recall, we asked them to use their browsing histories and original code bases to cue their retrospective walkthroughs. The interviews focused on three main questions: *(1) what types of resources were used and were thought to be the most useful; (2) what types of information the participants would like to keep track of; (3) how was information captured during the process, and were participants able to remember and effectively understand the information at a later time.*

B. Results

For question 1, we found that the typical go-to resources fell into five general categories: (a) community Q&A sites such as Stack Overflow and Quora; (b) official documentations of APIs and SDKs; (c) tutorial sites such as W3Schools, and TutorialsPoint; (d) blog posts such as those from Medium; and (e) human intelligence such as experts and co-workers. Of these, community Q&A sites like Stack Overflow were the most frequently visited, which is congruent with prior research [24], [25]. This motivated us to particularly optimize our system for Stack Overflow. With respect to human intelligence, one of the participants also reported using social media (Twitter) as an instant source of validation by tweeting about a suspected bug in the current Node.js implementation and waiting for others' confirmation online before filing a bug report. We found this behavior interesting and was not reported before in prior literature. This motivated UNAKITE to support collecting snippets from any web-based sources, even Twitter.

Question 2 was critical in informing the design of what information to capture and how to structure it in useful ways. Although developers consumed a large variety of types of information, we identified five different categories that seemed to describe the key ways they structured this information: (1) options or alternatives to solve the problem; (2) advantages and disadvantages of each option; (3) source URLs of web pages that contain relevant code snippets; (4) contexts and restrictions that relate to each option; and (5) indicators of which part of the information space has been explored and which has not. We designed UNAKITE to support capturing all five of these categories of information.

The last question investigates how developers currently keep track of design rationale. Different personal approaches to recording information were reported (reported next by the increasing amount of effort required of the developer): (a) keeping everything in their head without externalizing anything (7 participants did this. E.g. *"As soon as I got the answer, I just throw it in my code and move on. I don't document it for later."* – A12); (b) recording source URL in code comments (3 participants did this. E.g. *"Once I identify which bug caused [the problem], I just put that link to the bug report in my comments."* – A11); (c) taking notes in a note-taking app like the MacOS Notes application, or posting code snippets on code sharing websites such as CodePen or Github Gist (4 participants did this. E.g. *"I used to use Trello a lot."* – A7. *"I wrote a small piece of code using the Gist feature on Github and shared it with another developer."* – A1); (d) building custom personal note-taking systems (1 participant did this. *"I built a web app specifically for taking notes so that I could quickly refer back to them."* – A4). It is striking that almost half of the participants (7 out of 15) simply trusted their memory due to the high cost of both taking notes elsewhere and finding the relevant ones at a later stage (option a). Unfortunately, their memories were not as reliable as they thought they would be, resulting in repetitive searches for similar, if not the same, investigation tasks (*"If I need it later and I don't [remember], I just type it in Google and start over"* – A12). This suggested a potential opportunity for our system to provide developers with a better mechanism to record design rationale which is low-cost and seamless.

IV. UNAKITE IN ACTION

This section presents an example composite scenario that embodies many of the use cases identified in our preliminary studies. UNAKITE introduces a simple model that structures information into: (1) *options* (possible ways to solve the problem), (2) *requirements* (potential situations where the options would be considered good), and (3) *snippets* (relevant chunks of information on any web pages, some with sample code). In this scenario, we show how a developer could take

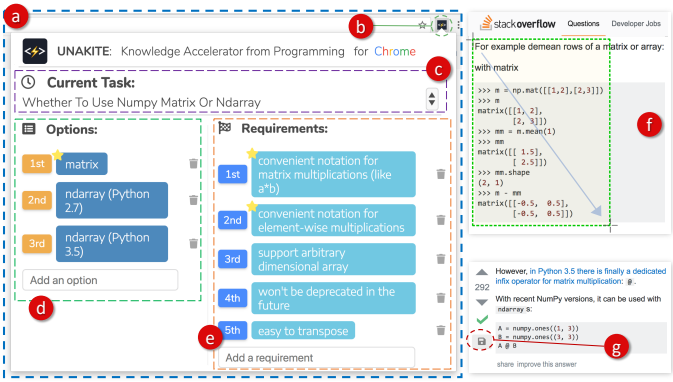

Fig. 2: Screenshots of UNAKITE Chrome extension: (a) Chrome extension popup menu; (b) extension icon (click it to show the *popup menu* (left)); (c) *task switcher* (current task name is editable); (d)-(e) draggable, sortable options & requirements list; (f) collecting a snippet by drawing a bounding box while holding the "Alt" key ("Option" key on MacOS); (g) collecting a snippet using the Stack Overflow dedicated clipping button.

advantage of this model to solve a fairly complicated research task.

Sara, a junior professional python developer, is trying to perform some matrix operations using `numpy` [26]. She immediately runs into the problem of how to represent matrices using `numpy`, as she had little prior experience using this package. She types in the search query "how to represent matrices in `numpy`" in her browser to start a Google search. UNAKITE will automatically create a new *task* with the search query as its name.

As Sara goes through the search results, she came across a Stack Overflow post about how to use `numpy matrix` to carry out simple matrix operations. She then adds `matrix` as an option from the UNAKITE Chrome extension popup menu (see Fig. 2a). (Alternatively, she could also select the word "matrix" on the web page and add it using the "Add 'matrix' as an Option" button from the Chrome context menu, which avoids typing.) Options are listed both in the popup menu (see Fig. 2d) and in the *comparison table view* (see Fig. 5c).

As Sara continues to read the post, she comes across several requirements that she would like to keep track of. For example, "having a convenient notation for matrix multiplication like a*b" is essential for the code to not be messy. Therefore, Sara adds them into UNAKITE using the same mechanism as if she is adding options (either from the popup menu shown in Fig. 2e or the context menu).

By going over the search results, Sara found that `ndarray` is another potential option. Since she saw some big differences between the usages of `ndarray` in the two major Python versions, she then adds `ndarray (Python 2.7)` and `ndarray (Python 3.5)` as two separate options. Now that she feels more confident about the popular options used to represent matrices using `numpy`, she decides to edit the task name in the popup menu (see Fig. 2c) since the task has effectively changed to be "whether to use `numpy matrix` or `ndarray`".

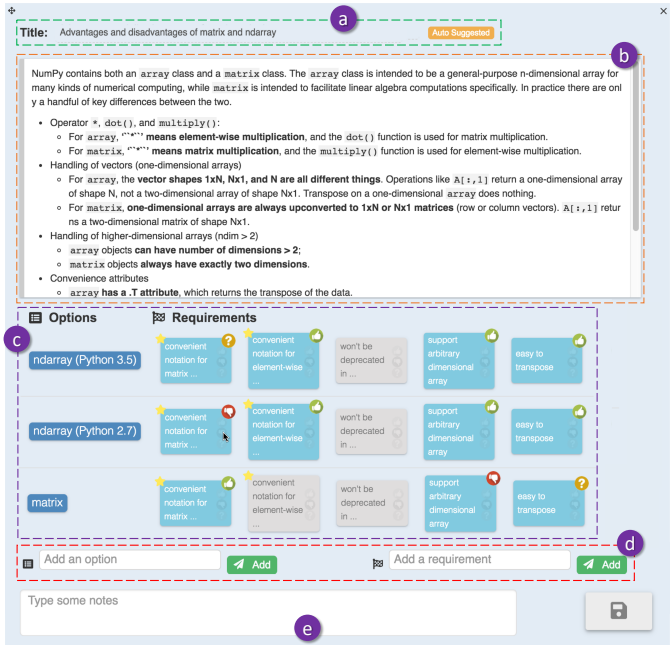

Fig. 3: UNAKITE Chrome extensions's *clipping interface*: (a) the title (editable) of the snippet; (b) the clipped content with the original HTML styling; (c) the *rating pane* where users can rate options against requirements; (d) where users can add options and requirements; (e) where users can take notes.

Sara then sees a section in a blog post that states the advantages and disadvantages of `numpy matrix` and `ndarray` that is worth keeping track of as a snippet. She uses the mouse to draw a bounding box around that chunk of information as shown in Fig. 2f. In the *clipping interface* (see Fig. 3) that pops up, Sara uses the information in the clipped content (see Fig. 3b) to fill out the table in the rating pane (see Fig. 3c). For example, she gives a "thumb up" (good/positive) in "having a convenient notation for matrix multiplication like a*b" for both the `ndarray (Python 2.7)` and `ndarray (Python 3.5)` as suggested in the clipped content. Alternatively, a user can also give "thumb down" (bad/negative) or "question mark" (not sure). She then saves the snippet by hitting the save button.

A *comparison table* (see Fig. 5) is automatically built up as Sara continues to collect information using UNAKITE. Sara periodically checks the table to gain a bigger picture of the information space and re-order the requirements and options according to their importance by dragging (see Fig. 5d). When it comes to decision making, Sara realizes that the company server she has access to only supports Python 2.7 (external constraint), and `numpy matrix` appears to be the best choice at the moment.

When Sara opens the code editor, she can access a *collection view* (see Fig. 4c) of all the snippets she collected or the *comparison table view* (see Fig. 5) in an editor side panel. She then copies a piece of sample code from one of the snippets, pastes it into her existing code, and modifies it to fit her needs. Behind the scenes, UNAKITE embeds a special

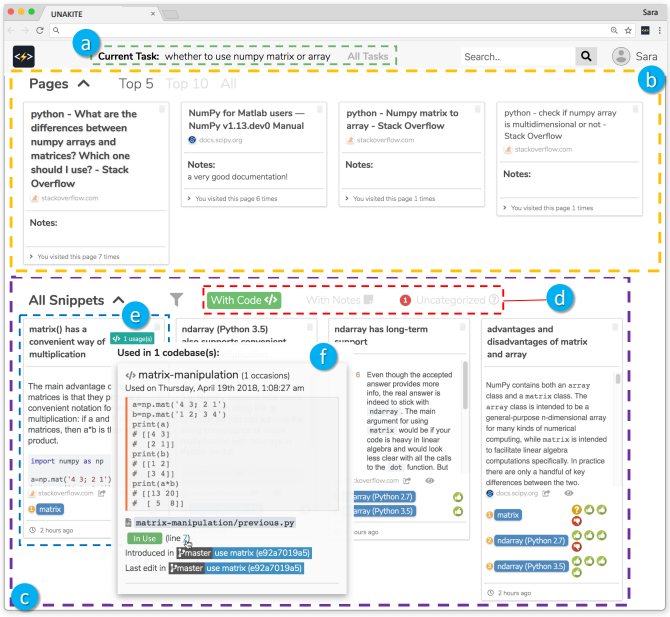

Fig. 4: UNAKITE's *collection view* of the *current task* (viewed in a new tab in Chrome): (a) *navigation bar* where users can navigate between the *current task* and the list of *all tasks*; (b) the browsing history specific to the *current task*; (c) the collection of all the *snippets*; (d) snippet filters; (e) an individual *snippet card*; (f) the code usage information of the selected snippet.

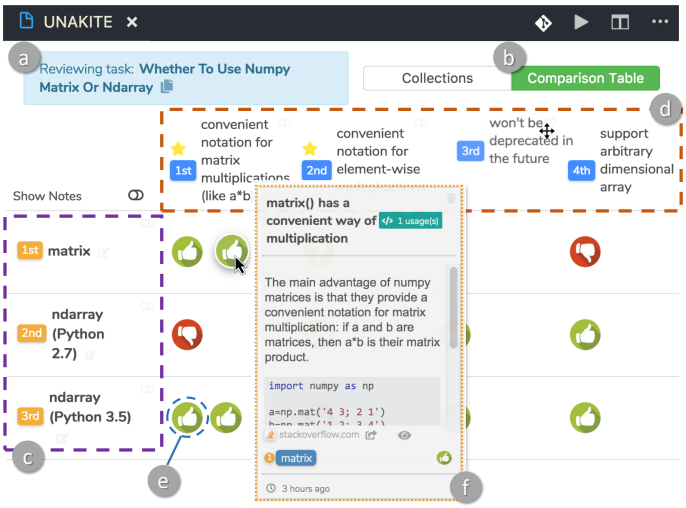

Fig. 5: UNAKITE's *comparison table view* of a task (viewed in a Visual Studio Code side panel): (a) the name of the task the developer is reviewing; (b) *collection view* & *comparison table view* switcher; (c) row header with the *options*; (d) column header with the *requirements*; (e) an individual *rating* that rates an option (`ndarray (Python 3.5)` in this case) against a requirement ("convenient notation for matrix multiplications (like a*b)" in this case); (f) the snippet card that provides supporting evidence for the selected rating (reveals when hovering the cursor over).

comment referencing the task that Sara generated as the design rationale into the project above the pasted code.

A year later, Larry comes in and reads the code that Sara wrote. He brings up a popup window that contains the essence of Sara's design rationale for using `numpy matrix` by hovering over the special comment (see Fig. 1a). Thanks to this, he then quickly understands Sara's decision back then, and realizes the opportunity to switch to `ndarray (Python 3.5)` since now the server supports Python 3.5 in addition to 2.7. He then opens the task from an editor panel to review and build on the full design rationale.

V. Details of The UNAKITE System

UNAKITE consists of an extension to the Chrome Web Browser and an extension to the Visual Studio Code editor. The browser plugin is implemented in HTML, Javascript and CSS, using the React Javascript library [27]. The editor extension is implemented in TypeScript according to the VS Code extensibility reference. Microsoft's Visual Studio Code is a popular cross-platform open-source code editor [28] (UNAKITE could be ported to other browsers and code editors). UNAKITE uses Google Firebase [29] database service for real-time communication and data persistence.

A. Capturing Knowledge

Creating a new Google search will trigger UNAKITE to initiate a new task. All subsequent actions developers perform and the knowledge they collected will be associated with this task by default. We specifically designed this behavior to eliminate the friction of starting to use UNAKITE to keep track of information. Since people often use query reformulations as they narrow down to the search target [11], [30], UNAKITE additionally provides many features to support editing tasks and preventing "task explosions" (inadvertently creating too many tasks): users can easily discard tasks that are no longer useful or drag one task over another to combine them in the *task browser* (discussed below). Switching among tasks can also be easily done through the *task switcher* in the popup menu (see Fig. 2c).

UNAKITE provides multiple ways to add options and requirements. In addition to what was described in the sample scenario, the user could also add them from the clipping interface (see Fig. 3d) in one operation, e.g., when she finds a new snippet specifically about a new option and a new requirement. In cases where a user has some prior knowledge of what options and/or requirements are to be considered, she could use the Chrome extension popup menu (see Fig. 2d&e) to quickly add a batch of them.

When a user collects a snippet by drawing a bounding box, unlike in previous tools [30], [31] where information was saved in pure text format, UNAKITE is able to show the content with its original styling and include the rich, interactive multimedia objects supported by HTML, including images and links. This feature helps developers quickly recognize a particular snippet among many others in the collection view (see Fig. 4c) if they remember how it looks. UNAKITE also allows users to add titles (see Fig. 3a) and notes (see Fig. 3e) to a snippet. The same clipping interface can be used to edit snippets at a later stage.

There are also multiple ways to collect snippets in UN-

AKITE. In addition to drawing a bounding box, users could also select relevant text and bring up the clipping interface (see Fig. 3) from the context menu. Since Stack Overflow is the most frequently visited community Q&A website by developers [13], [25], UNAKITE also offers two additional ways of collecting snippets from it: (a) click the clipping button we insert into the Stack Overflow pages (see Fig. 2g), which saves the post as a snippet; (2) copy-and-paste a piece of sample code from Stack Overflow directly into the user's existing code in the Visual Studio editor, which also creates a snippet containing the code and the information within its vicinity. Thus, the starting cost for developers to use UNAKITE is minimized, and users have the flexibility to collect knowledge in a variety of ways and orders, without needing to follow a preset pattern.

B. Organizing Knowledge

UNAKITE adds a new tab to Chrome that runs an interactive web app through which developers can access all the information they collected. It provides the collection view (see Fig. 4) and the comparison table view (see Fig. 5) where users can organize the collected knowledge, as well as a task browser (accessible through clicking "All Tasks" in the navigation bar shown in Fig. 4a) where users can navigate to any past task to view, edit, star or discard it.

The collection view is the default way to view a task. It serves as a comprehensive dashboard that holds a collection of all the collected snippets (see Fig. 4c) in the form of *snippet cards* (see Fig. 4e), as well as a browsing history (see Fig. 4b) associated with the current task. Imagine that Sara wants to go back to a specific snippet that was collected to copy the code, she could search (find) the snippet if she remembers any text in it, or else quickly browse through the snippets looking at the titles of the snippets or the visual cue provided by the content preview. If there are too many snippet cards, Sara could turn on the "With Code" filter to only see snippets that contain code, or the "With Notes" filter to only see snippets that she wrote note for (see Fig. 4d).

A key advantage of UNAKITE is the comparison table view that is automatically created as the user collects the options, requirements, and snippets. It compares the options vs. the requirements, with each cell occupied by an aggregated list of ratings from all the relevant snippets. By hovering the cursor over each rating, the relevant snippet will be revealed as evidence.

There are two major use cases that the comparison table is designed to support. First, it offers users a big picture of their research progress and helps them understand what part of the information space has been explored, as discussed above as the fifth requirement for knowledge support. This provides developers with clues about to which branch they might want to redirect their subsequent search efforts, e.g., if there's evidence shown that both both `ndarray (Python 2.7)` and `ndarray (Python 3.5)` are good in terms of "will not be deprecated in the foreseeable future" but the cell corresponding to `matrix` is still empty, Sara might want to specifically start looking out for evidence to fill in this blank.

Secondly, the table acts as a space in which developers can organize their thoughts as they are trying to make a decision, and then using the results of this organization as the design rationale for the final decision, as shown in the scenario. In addition, UNAKITE also provides support to "star" an option or a requirement if it is crucial, "mask" it if it is no longer valuable for the task (see Fig. 5c&d), and "decide" to choose an option by clicking the green checkmark (Fig. 6b).

UNAKITE uses the up/down/don't-know model to capture the relationship between options and requirements even though in some cases such a relationship will not necessarily be easy to express. We specifically chose this design to make it quick and easy for developers to generate and edit ratings. If the developer wants to represent more nuanced decisions, they can always attach notes to the options explaining the issues. In the future, we will consider adding support for textual and numerical ratings to capture more complex cases.

Another design choice we made was to prevent developers from directly inputting ratings into the comparison table according to their personal beliefs, because this would circumvent the entire process of collecting snippets as evidence to back their decisions. For example, if they were wrong about their prior beliefs, it could lead to unverified ratings, untenable decisions, and eventually a untrustworthy design rationale. The current workflow of UNAKITE, to some extent, forces developers to look for evidence as they build up their confidence in making a final decision, and in turn makes the generated design rationale more trustworthy.

C. Persisting Knowledge

Prior work has identified that one of the biggest obstacles in keeping track of design rationale in software projects is the difficulty of integrating design rationale systems into developers' normal workflow [5]. This is, however, under the assumption that design rationale is to be "summarized" *after* the decision has been made, with additional effort required from the author to externalize the thought processes. UNAKITE breaks this deadlock by moving the externalization upfront when developers are formulating their decisions (discussed above). These externalized artifacts are retained to help the developer and possibly future developers see what options and requirements were considered, and which sources were found useful. As shown above, the Visual Studio Code editor extension can seamlessly embed this generated design rationale into the code.

UNAKITE also makes these connections if the user copies and pastes the code from the UNAKITE web app in Chrome instead of from the editor panel, or even if the code is copy-and-pasted from Stack Overflow directly into the code editor. A manual connection can also be made between a point in the code and information in the web app without copying and pasting any code, for example for decisions like which API to use, which might not immediately involve any copy-and-paste.

We want to stress that this automatic embedding of design rationale fits into developers' normal workflow of copying and pasting code without requiring any extra effort. Developers are also free to change the pasted-in code any way they want without messing up the embedded design rationale.

D. Exploring Design Rationale

When the user makes a connection between the web app and the code, UNAKITE goes significantly beyond prior work [9], [12] where only the source URLs of the web pages were recorded, by linking to the full design rationale as discussed above. There are many ways the user can view and explore this connection:

1) Special comment and popup window: Once a connection is established, UNAKITE will anchor the design rationale by placing a special comment containing a unique id in the code. By hovering over the comment, UNAKITE will present a popup window that contains the key parts of the design rationale (see Fig. 1b), including the task name, the complete comparison table, the snippet title, options, requirements, ratings and notes specific to the relevant snippet, and the originally copied code. This is designed to offer developers a low-cost way to see why the decision was made without opening up the full rationale in the web app. This form of richer annotations was also confirmed to be more valuable than plaintext comments by prior research [32].

2) From code to UNAKITE *web app:* Users could also navigate to the full design rationale by clicking on the task name or the snippet title in the popup window. This will bring up the UNAKITE web app in an editor panel (see Fig. 5), in which the relevant snippet is marked by an orange dotted border. Unlike traditional methods that require manually opening up the source URL in a browser [12], UNAKITE directly presents the web app as if the user is checking another code file, which reduces the number of context switches and boosts the user's work efficiency.

3) From UNAKITE *web app to code uses:* The user can see all the usages of a piece of code in a snippet across all of the different projects (see Fig. 4f) from the web app. This information is automatically kept up-to-date even if any edits happen. Developers can navigate to each usage instance simply by clicking the corresponding line number in the report. We designed this feature to provide developers with a clear picture of when and where a piece of code was used, and greatly reduce the effort of code searches [11]. Inspired by Deep intellisense [33] and Chronicler [34] that provide developers with code-snippet level history, UNAKITE also integrates Git and provides a brief longitudinal insight for each usage instance including when (in which commit) the sample code was first introduced into the code base, when was it modified, and possibility when was it taken out (as shown in Fig. 4f), which also contribute to understanding the design rationale as suggested by prior research [1], [3], [4].

VI. PRELIMINARY USABILITY STUDY

To test how successfully users can interact with some features of UNAKITE, we conducted a pilot usability study

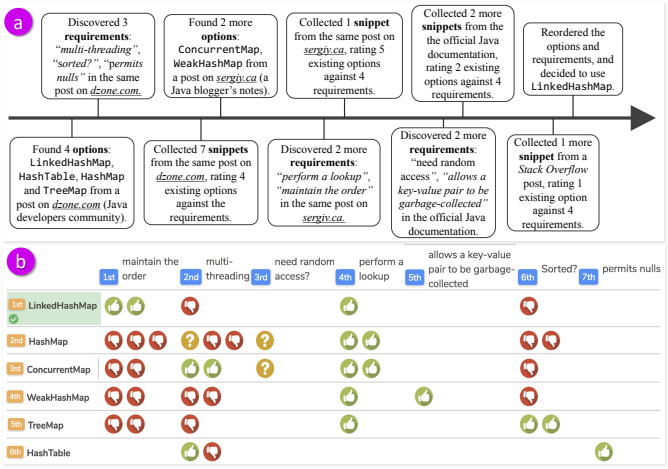

Fig. 6: Participant P2's interactions with UNAKITE: (a) a brief replay of P2's research process (only key actions related to *option*, *requirement*, *snippet*, and *decision-making* are included); (b) the resulting comparison table.

where we evaluated developers' ability to use UNAKITE to help with their decision-making processes and their ability to comprehend previously-generated design rationales.

We recruited 5 participants from the local university community, all of whom were doctoral students with an undergraduate degree in computer science (3 male, 2 female). Participants had on average 8 years of programming experience, with the longest being around 20 years. The pilot study was conducted in a lab setting, using a designated MacBook Pro computer with the latest version of Chrome, Visual Studio Code and UNAKITE installed. After signing the consent form and receiving a tutorial on how to use UNAKITE, each participant was asked to finish two tasks: *(1) researching with* UNAKITE and *(2) understanding design rationale*. After finishing the tasks, each participant filled out an online survey to give feedback about UNAKITE, and was then compensated $20 for their time.

A. Researching with UNAKITE

Each participant was asked to research using UNAKITE for 15 minutes on a programming task of their choosing. This evaluated the developers' ability to understand and use the features of UNAKITE, and more importantly, how UNAKITE fits into their workflow.

In general, all participants were able to successfully use UNAKITE to discover new options and requirements, collect snippets as evidence, and finally make a decision. For example P2 identified 6 available options and 7 different requirements, and collected 11 snippets while investigating the task "choose map implementations in Java". Fig. 6 shows a brief replay of his research process as well as the resulting comparison table.

All participants reported in the survey that they liked the experience of using UNAKITE as a research companion. P3, who researched "set up website using JavaScript front-end [framework]", expressed a strong desire to have UNAKITE help with a variety of tasks: *"The interface is very clean, simple,*

and intuitive. I can definitely see myself using it virtually for all my programming tasks and staying organized." P5, who researched "JavaScript render gif on canvas", liked the fact that the original styling is preserved in collected snippets: *"I really loved the fact that the styles of the snippets were exactly as they were in the cards so that I could preview it and quickly identify them."* P4, who researched "static site generator [using] React", appreciated the big picture that the comparison table provides: *"I really liked this table. I think at this point it makes clear for me that I should look into the size of the community of different projects. It's like I know what I'm missing."*, and also suggested its value for other programmers on the originating web site as well: *"I feel like if you post this [the comparison table] to Stack Overflow, people would definitely appreciate it."*

As recommendations for improvements, P3 pointed out that collecting a single snippet could involve up to 6 mouse clicks, which could be a burden. This motivated some of the features later added to the system, including the dedicated clipping buttons on Stack Overflow and the implicit collection of snippets upon copying and pasting. P1, P3 and P5 wished that UNAKITE supported team collaborations. This motivated us to support the "clone task" feature, where developers could clone existing tasks to their own task repositories, refine them, and share them later. Section VII will further address this issue.

B. Understanding Design Rationale

Each participant was also asked to interpret an already existing design rationale generated with UNAKITE. Participants were initially provided with a piece of Python code that does matrix multiplications with embedded design rationale as shown in Fig. 1. The code and the design rationale were generated by the author based on Sara's scenario described in section IV. Each participant was asked to explain why the original decision was to use `matrix` given the constraints and whether to make a different decision given that the company now supports Python 3.5. It was designed to evaluate developers' ability to comprehend design rationales generated by previous developers as represented by UNAKITE.

All participants successfully explained the decision with the help of the design rationale. P1 and P5 figured out the answer with just the information in the popup window (see Fig. 1), while the other 3 participants reviewed the original task in the side panel (see Fig. 5). All participants also came up with the better solution using `ndarray (Python 3.5)` along with the existing sample code in the task when the constraints changed.

When asked about the role of UNAKITE in helping them understand the design decision, all participants stated that they initially had no clue about why `matrix` was used instead of other possible options by just looking at the code itself. They confirmed that the comparison table in UNAKITE helped them realized that `matrix` was indeed the best option given the order of importance of the requirements and the constraints. This provides evidence that developers are able to understand the design rationales presented by UNAKITE, and the design rationales could indeed help developers interpret previously-made design decisions quickly and successfully.

VII. FUTURE WORK

For future development, we plan to make UNAKITE smarter and more comprehensive. We plan to apply some lightweight machine learning methods to help determine which tasks, options and requirements that snippets go with. We will adopt some basic natural language processing techniques like machine summarization and entity extraction to intelligently fill out the rating pane as well as the snippet title in the clipping interface after analyzing the clipped content. We would also like to take a step further in helping developers make decisions by recommending the optimal decision based on the content of the comparison table. We will also extend UNAKITE to help provide design rationale for other kinds of decisions beyond ones involving choosing among options.

In addition, we would also like to explore the possibility of making UNAKITE a collaborative platform. Building on the current "clone task" feature , we plan to support GoogleDoc style collaboration where multiple developers could collaborate on the same task simultaneously. This will involve an easy way to split the workload among different users, a robust synchronization mechanism, and a built-in real-time communication system like Slack [35]. We also plan to enable developers to refine a piece of design rationale to lower the burden of decision-making by later developers, especially for those who are facing the same task.

An obvious next step is to evaluate UNAKITE's usefulness in the field. We are preparing the system for a long-term usage by both professional and novice developers. We are hoping to see how UNAKITE will be used in everyday workflow over a period of weeks or months. We would also like to explore the possibility of making UNAKITE an intervention mechanism to promote a structured way of solving coding problems and help developers form the habit of staying organized. Eventually we plan to release UNAKITE as an open source project.

VIII. CONCLUSION

UNAKITE supports developers in collecting, organizing, consuming, and persisting design rationales for choosing among options while solving problems using web resources. We designed and iterated UNAKITE based on the data we collected from our formative investigation, ideas and issues suggested by prior work, and usability feedback from software developers. UNAKITE addresses the critical needs of (1) collecting and organizing information and (2) keeping track of the associated design rationales. Pilot usability study showed that the system is usable, effective and desirable.

Capturing design rationale is well-known for being a challenging task in the software engineering community. UNAKITE suggests a possible systematic approach to go from initial knowledge gathering to the final embedding of design rationales in code with the maximum amount of cognitive assistance and the minimum amount of mental overhead.

REFERENCES

[1] T. D. LaToza and B. A. Myers, "Hard to Answer Questions about Code," in *Second Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'2010) at SPLASH/Onward! 2010*, 2010.

[2] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett, "Foraging and Navigations, Fundamentally: Developers' Predictions of Value and Cost," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 97–108.

[3] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM, May 2010, pp. 175–184.

[4] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, May 2007, pp. 344–353.

[5] A. P. J. Jarczyk, P. Loffler, and F. M. Shipmann, "Design rationale for software engineering: a survey," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*. IEEE, 1992, pp. 577–586 vol.2.

[6] V. Arora, J. E. Greer, and J. P. Tremblay, "A framework for capturing design rationale using granularity hierarchies," in *[1992] Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*. IEEE Comput. Soc. Press, pp. 246–251.

[7] T. P. Moran and J. M. Carroll, Eds., *Design Rationale: Concepts, Techniques, and Use*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1996.

[8] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *SIGSOFT'06/FSE-14: Proceedings of the 13th ACM SIGSOFT and 14th international symposium on Foundations of Software Engineering*, 2006, pp. 23–34.

[9] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," in *Visual Languages and Human-Centric Computing (VL/HCC'06)*, Sep. 2006, pp. 195–202.

[10] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. D. Penta, and M. Lanza, "Supporting Software Developers with a Holistic Recommender System," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 94–105.

[11] C. Sadowski, K. T. Stolee, and S. Elbaum, "How Developers Search for Code: A Case Study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 191–201.

[12] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-Centric Programming: Integrating Web Search into the Development Environment," in *CHI'2010: ACM Conference on Human Factors in Computing Systems*, 2010, pp. 513–522.

[13] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack Overflow in the IDE," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1295–1298.

[14] Y. Chen, S. W. Lee, Y. Xie, Y. Yang, W. S. Lasecki, and S. Oney, "Codeon: On-Demand Software Development Assistance," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: ACM, 2017, pp. 6220–6231.

[15] S. P. Davies, "Externalising Information During Coding Activities: Effects of Expertise, Environment and Task," in *Empirical Studies of Programmers: Fifth Workshop*, C. R. Cook, J. C. Scholtz, and J. C. Spohrer, Eds. Palo Alto, CA: Ablex Publishing Corporation, 1993, pp. 42–61.

[16] W. Kunz and H. W. J. Rittel, *Issues as elements of information systems*. Institute of Urban and Regional Development, University of California Berkeley, California, 1970, vol. 131.

[17] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the International Conference on Software Engineering (ICSE'2006)*, 2006, pp. 492–501.

[18] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *ESEC/FSE 2007: ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007, pp. 361–370.

[19] B. Fluri, M. Wursch, and H. C. Gall, "Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, Oct. 2007, pp. 70–79.

[20] T. Tenny, "Program readability: procedures versus comments," *IIEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1271–1279, 1988.

[21] M. L. Van De Vanter, "The documentary structure of source code," *Information and Software Technology*, vol. 44, no. 13, pp. 767–782, Oct. 2002.

[22] B. Fluri, M. Wrsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, Dec. 2009.

[23] M. Mahoney, "Collaborative Software Development Through Reflection and Storytelling," in *Companion of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. ACM, Feb. 2017, pp. 13–16.

[24] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, "Answering questions about unanswered questions of stack overflow," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, May 2013, pp. 97–100.

[25] A. Fourney and M. R. Morris, *Enhancing Technical Q&A Forums with CiteHistory (PDF Download Available)*, 2013. [Online]. Available: https://www.researchgate.net/publication/262068976_Enhancing_Technical_QA_Forums_with_CiteHistory

[26] *NumPy NumPy*. [Online]. Available: http://www.numpy.org/

[27] Facebook, *React - A JavaScript library for building user interfaces*. [Online]. Available: https://reactjs.org/

[28] *Stack Overflow Developer Survey 2018*. [Online]. Available: https://insights.stackoverflow.com/survey/2018/

[29] *Firebase*. [Online]. Available: https://firebase.google.com/

[30] A. Kittur, A. M. Peters, A. Diriye, T. Telang, and M. R. Bove, "Costs and Benefits of Structured Information Foraging," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 2989–2998.

[31] A. Kittur, A. M. Peters, A. Diriye, and M. Bove, "Standing on the Schemas of Giants: Socially Augmented Information Foraging," in *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ser. CSCW '14. New York, NY, USA: ACM, 2014, pp. 999–1010.

[32] A. J. Ko and B. A. Myers, "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 387–396.

[33] R. Holmes and A. Begel, "Deep intellisense: a tool for rehydrating evaporated information," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, May 2008, pp. 23–26.

[34] M. Wittenhagen, C. Cherek, and J. Borchers, "Chronicler: Interactive Exploration of Source Code History," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, May 2016, pp. 3522–3532.

[35] Slack, *Where work happens*. [Online]. Available: https://slack.com/