Pontem: Improving Developer Program Comprehension Through Code Tagging

Patrick Boutet
Department of Computer Science
The University of British Columbia
Vancouver, British Columbia, Canada
pboutet@cs.ubc.ca

Kristian Flatheim Jensen
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Trøndelag, Norway
krisfjen@stud.ntnu.no

Lucas Palazzi
Electrical & Computer Engineering
The University of British Columbia
Vancouver, British Columbia, Canada
lpalazzi@ece.ubc.ca

ABSTRACT

Software developers often face problems with program comprehension. Software documentation is a crucial component in maintaining a well-organized and comprehensible codebase, and as such it is important that developers have useful and effortless forms of documentation. While code-level comments offer a quick way of documenting the functionality of code, it does not provide much use for code organization. On the other hand, external documentation can be more useful for providing information related to code organization and structure, but it is often difficult and time-consuming to maintain.

In this paper, we introduce a model that allows developers to tag sections of code with user-defined labels in order to help place that code in the broader context of the codebase. This additional layer of documentation would help developers better comprehend their code while still maintaining the simplicity of in-editor documentation. Our code tagging model allows users to define "Tags" that can be applied to multiple code segments in the form of "Instances". Developers can then navigate their codebase based on a specified Tag in order to better understand a specific component in the program.

We implement our code tagging model in the form of an extension for Visual Studio Code. Our code tagging tool, named Pontem, allows developers to tag segments of their code with user-defined tag names, navigate their codebase based on tags, and visualize the tag distribution in their development environment. Our tool was evaluated using qualitative interactive demos and interview sessions. The feedback received during our evaluation was generally positive, suggesting that our code tagging model has potential.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**; **Software notations and tools**; *Software maintenance tools*; *Software development process management*; Collaboration in software development;

ACM Reference format:
Patrick Boutet, Kristian Flatheim Jensen, and Lucas Palazzi. 2018. Pontem: Improving Developer Program Comprehension Through Code Tagging. In *Proceedings of CS507 - Human Aspects of Software Engineering, Vancouver, BC, April 2018 (CS507'18),* 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CS507'18, April 2018, Vancouver, BC
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the software development industry, there is an ever-present desire to have better code comprehension among both new and existing team members [20]. New developers often face issues when first exposed to an existing codebase, such as knowing what different sections of the code do and knowing who to ask when facing a problem [8]. For existing team members, navigating based on code functionality and keeping track of who has worked on different features can be challenging [5, 6]. Typically, this is accomplished by maintaining a set of comprehensive documentation to accompany the codebase.

While current forms of documentation provide useful information to developers regarding how a software system operates, they are limited in terms of the benefits that they provide. For example, code-level comments are effective at giving programmers information on how a specific code segment functions without having to leave their development environment. Also, code-level comments are typically easy to keep up-to-date since they are short and "right there" [10]. However, comments are not typically used for code organization, such as keeping track of where in the codebase different features are implemented, as it can be onerous to search and filter the codebase for comments containing such information. On the other hand, external documentation can be more useful for providing such code organization information, but in order to access this information a developer must leave their development environment, search for the relevant information, and trace it back to the source code. This context switching can be cumbersome and time-consuming. In addition, it requires more effort to keep external documentation up-to-date compared to code-level comments that are embedded in the source code [10]. In general, it is expensive for developers to context switch and more so if they have to change program [12]. Therefore increasing the amount of organizational information that a developer sees without leaving their environment could be very beneficial.

This paper introduces a model for tagging sections of code with user-defined tags to provide better program comprehension. Our code tagging model provides an additional layer of documentation that would allow programmers to better organize their code and keep track of which files and code blocks are tagged with certain information. We discuss potential components and features, such as the information contained in a tag, the visualization of tags, and the ability to navigate a codebase based on tags. We provide an implementation of this model in the form of an extension for Visual Studio Code, a widely-used open-source text editor. Our code tagging tool, named Pontem, allows developers to tag segments of

their code with user-defined tag names, navigate their codebase based on tags, and visualize the tags in their development environment. We validated our code tagging model through a study in which 4 programmers, equipped with Pontem, navigated a provided codebase and completed some small tasks. A short interview was conducted with each subject after using the tool. The results of our study showed that our code tagging model has promise, with mostly positive feedback from the participants.

Section 2 presents the motivation behind our code tagging model. Section 3 provides an overview of related works. In Section 4, we present a generic description of our code tagging model, including its components and features that would help developers better organize their codebase and improve comprehension and awareness. In Section 5, we describe how this model is implemented as a text editor tool. Section 6 describes the experimental setup for our evaluation, as well as the research questions we sought to answer. Our results are presented in Section 7, which are then discussed in Section 8. Section 9 provides the threats to the validity of our evaluation. Finally, we conclude the paper in Section 10 by discussing possibilities for future work on the ideas presented in this paper and areas of improvement for our tool.

2 MOTIVATION

Motivation for our model and tool implementation originated from the program comprehension problem that many new developers face when joining an existing software development project. First, we know that on a general level developers typically have many information needs, such as awareness about artifacts, needing to know about design and program behavior, and understanding what a program component is supposed to do [8]. We also know that many new software developers struggle when it comes to cognition and orientation in low information environments, which are common in industry [1]. This brings us to the problem of program comprehension in the context of new developers joining an existing software development project. Recent studies show that industry developers spend approximately 58% of their time on program comprehension activities and that junior developers have a larger portion of their time devoted to these activities than senior developers [20].

As developers, we are familiar with the challenges and frustrations associated with being a new member of a software development project and attempting to comprehend what the program does. With that in mind, we hypothesized that if a codebase was organized based on program features, a new developer navigating the code based on this model could much more easily comprehend the program.

This lead us to propose our code tagging model for developer code comprehension and an implementation of this model, named Pontem. That allows a codebase to be tagged as per a user defined mental model and navigated via an easy to use dashboard.

3 RELATED WORKS

With the considerable amount of time that goes into code comprehension, there is a large amount of research regarding the topic. Most recently a study by Xia et al. took a large industry level quantitative look at program comprehension, finding that approximately 58% of a developers time is spent performing program comprehension activities [20].This is one of the first studies that looked heavily at comprehension in industry and at comprehension activities across multiple applications, not just within an IDE. They also found that with regards to multiple applications, developers frequented web browsers and external artifacts during comprehension activities. An observations was that developers perform what they call "information foraging", which is a practice not well supported by current IDE's.

A publication by Maalej et al. examined a smaller cohort of professional developers, focusing on understanding their comprehension behaviour, strategies, and tools used [11]. They found that most developers followed pragmatic comprehension strategies, which were context dependent; that comprehension was avoided when possible; and that developers try to view programs from a users perspective through inspecting GUIs. A surprising finding was that there was almost no use of comprehension tools, suggesting that there is a gap between research and practice, and that changing of research agendas may be in order.

Two papers by Minelli et al. look at quantifying how developers spend their time, one in regards to program comprehension and the other at a general level [13, 14]. They find that from both studies, a large amount of a developer's time is spent on comprehension and that it has been typically underestimated in previous research. Since their work was on interaction data, they also propose an inference model to determine the type of activity: editing, navigating, searching for artifacts, interacting with the IDE, or corollary activities.

Taking a step back to older research, we still see the early trends in comprehension that we see today, only with less quantitative findings and under represented developer time measurements. Ko et al. performed an exploratory study regarding developers and maintenance tasks, showing that 35% of a developers time is spent on navigating and understanding code prior to editing [9].

A very unique 2005 review paper by Storey gives a lot of insight into the cognitive theories regarding program comprehension at that time [17]. It provides a comprehensive view of the landscape regarding the current research into program comprehension and suggests how future tools can be guided by cognitive theories regarding the way in which developers understand code. This leads into a 2006 paper where she proposes the initial concept of a shared way-point and social tagging tool called TagSEA [18]. The full implementation and evaluation of the tool is covered in her 2009 paper that investigates the use of code tagging by software developers to support reminding and re-finding [19]. TagSEA enables developers to tag locations in code with the goal of improving software navigation. They also presented results from two empirical studies where they observed how professional software developers use source code annotations in the development process. However, the intended use of their tool was more focused on LOC navigation, as opposed to a feature-based navigation. While the motivation for their tagging tool is slightly different than what we are proposing, the paper offers valuable insight into the need for such tools.

With the complexity that is associated with software development research, it is easy to find overlap within its many areas of focus. The concept of program comprehension encompasses many

different activities, and we find that other areas of research such as documentation or information needs have ties to comprehension.

A paper by Lethbridge et al. explores how software developers use documentation [10]. They found that software engineers rarely update and maintain documentation, and when they do it is usually several weeks after changes were made to the code. They also found that developers are more likely to update documentation when it is attached to the process of developing code, and not in a separate location. They stressed the need for simple, powerful, and easier to update documentation formats and tools. With comprehension relying on documentation as one of its underlying activities, it is easy to see that tagging tools could act as a proxy for documentation, potentially improving a projects overall documentation level.

In a paper by Ko et al. they investigates the information needs of software developers, and produced a list of information needs found in their observations [8]. They found that the information needs that frequently went unsatisfied were related to program behavior, such as "What code caused this program state?", "What code could have caused this behavior?", and "What is the program supposed to do?". These unsatisfied needs were found to clearly hinder developer productivity. Again we see overlap as these information needs seem similar to a comprehension problem, therefore developer productivity could be supported through the use of lightweight tagging tools that better abstract the intended purpose source code.

4 CODE TAGGING MODEL

4.1 Approach

We approach our model design in two stages. First, we utilize cognitive theories of program comprehension from early research in the field in order to help us understand how developers understand programs. We learned that while experienced developers look at programs from a top-down approach, new developers tend to look at programs from a bottom-up approach [17]. We reason that new developers could therefore benefit from the ability to also view a program from a top-down perspective when first exposed to a new codebase. Our code tagging model should therefore present a developer with a contextual representation of a codebase that gives a top-down perspective of the system.

Next, we considered findings from recent studies that look at program comprehension from a quantitative industry standpoint. When looking at the activities that developers perform during program comprehension, we learned that poor comments, program elements with no apparent meaning, and unfamiliarity with business logic were some of the reasons developers spend large amounts of time comprehending code [20]. To address these problems, we present some potential areas of improvement. Perhaps the underlying concept behind comments can be expanded to hold more information than it is currently capable of, thereby providing additional layers of comprehension to the developer. Also, while specific code elements might not always have an apparent meaning, if they can be categorized as part of a larger component that can be much more meaningful to a developer attempting to understand the code. This concept also addresses the business logic unfamiliarity issue, as business logic is analogous to a feature of a program; it is an abstracted level of functionality that the program needs to perform.

4.2 Model Definition

Our code tagging model presents a high-level overview of a codebase, while giving developers the freedom to decide how their codebase is defined. Under our model, a developer has the ability to tag sections of code, such as a classes, functions, or expressions, in order to label it as being a part of a larger component in the codebase. How these labels are defined is up to the user, which gives them the flexibility to use the model in the most appropriate way for the given programs context.

4.2.1 Definitions. We call a user-defined label a "Tag". We also define an "Instance" as a specific use of a Tag applied to a selection of code, containing additional metadata specific to that code segment, so a user can categorize multiple code segments (Instances) under the same label (Tag). The Instance abstraction level makes it possible for a code segment to be associated with multiple Instances of different Tags. This concept provides the core idea of our model.

4.2.2 Tag Creation. Under our model, a user can create a Tag by supplying a name that describes the program component it is to represent. Examples of possible Tag names include API, Database, User Login, To-do, and Bug. However, the user is given complete freedom to define the Tag names dependent on the unique components of their program. As a user is working within their codebase, they can create Instances of the Tags they created by invoking a command (keyboard shortcut, context menu item, or command pallet) and tagging selections of code.

4.2.3 Visual Representation. Once a selection of code is tagged as an Instance, the developer's workspace can provide a visual indication for the tagged code. In the user's text editor, any tagged code is highlighted using a colour that is unique to that Tag, ensuring that the user can easily distinguish between different components of the codebase.

4.2.4 Dashboard. Our model also includes support for a visual navigation tool in the form of a Dashboard. It presents a structured, organized representation of the program components and where the sections of code are that implement those components. The Dashboard can be used as an aggregation tool to visualize the Tags and Instances in a codebase. A well implemented dashboard should present a structured, organized representation of the program components and where the sections of code are that implement those components.

4.2.5 Tag Isolation. Our model supports isolating the files and source code of a project by Tag. When a user selects a Tag to isolate, our model does two things. First, it condenses the code that is open in the editor to focus only on the isolated Tag. It does so by collapsing all code elements such as functions and components that do not contain Instances of that specific Tag. No code is removed from the editor, as the user can still expand the collapsed code, but code that is not related to the selected Tag is no longer in the way. The second action involves condensing the source code directory to only focus on files that contain Instances of the isolated Tag. This can be done by either dimming all non-relevant files from the source tree so that they are not as prominent but still visible, or by hiding them completely from the tree. The former option is assumed to be preferable for most situations, as it allows the

developer to quickly distinguish between the relevant files while still being able to see and interact with the non-relevant files.

5 IMPLEMENTATION

Pontem [2] is an implementation of the model described in Section 4. It is created as an extension to Visual Studio Code [3] and enables the user to tag code. Pontem is created to be seamlessly added to an existing or new project and supports team collaboration.

5.1 Functionality

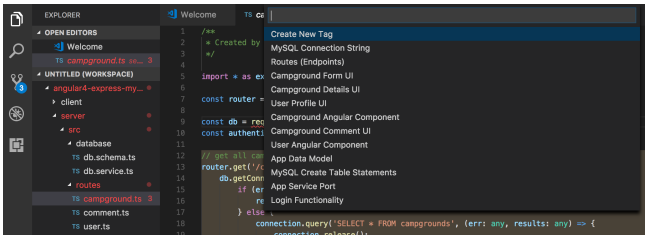

Figure 1: Creation of a Tag or an Instance is done through the QuickPick menu in Visual Studio Code.

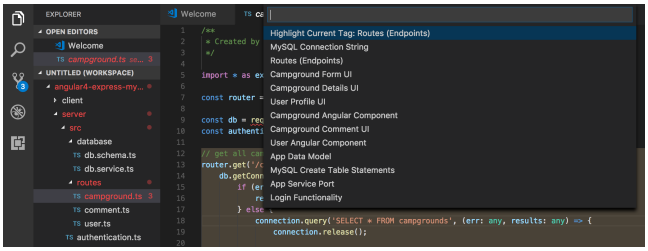

Figure 2: Entering isolation mode for a Tag is done through the QuickPick menu in Visual Studio Code.

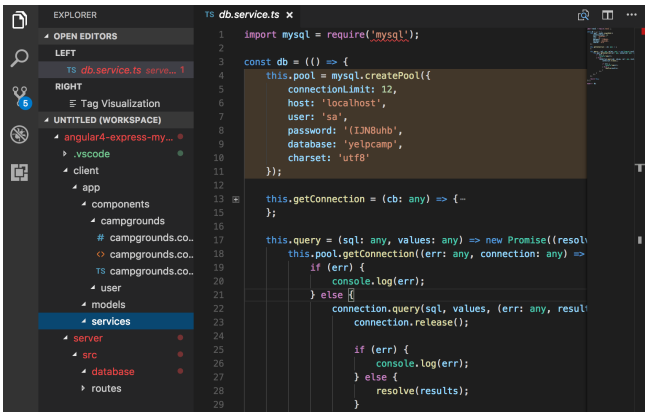

Figure 3: Isolation mode for a Tag automatically hides files without the Tag as well as folds functions/methods which do not contain an Instance of the Tag.

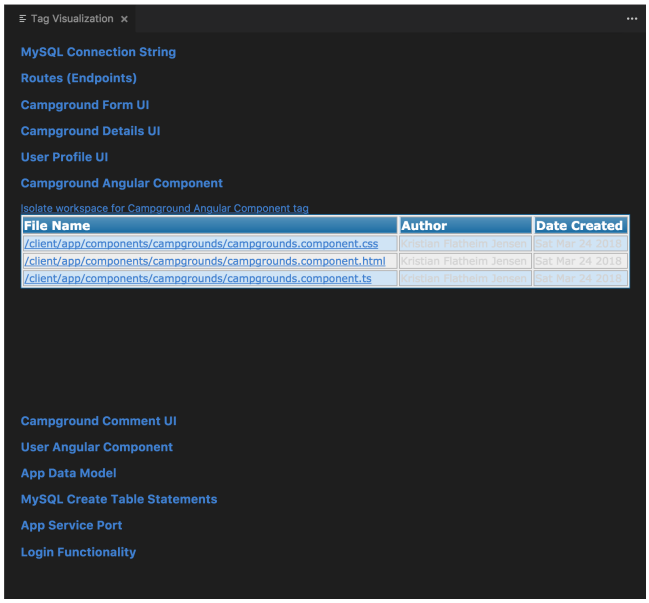

Figure 4: Dashboard shows all Tags in the workspace. Expanding a Tag shows the different Instances corresponding to the Tag as well as their location, author and date. There is also a button under each Tag to quickly isolate it. Clicking the filename of an Instance opens that file and jumps to the code segment of the Instance.

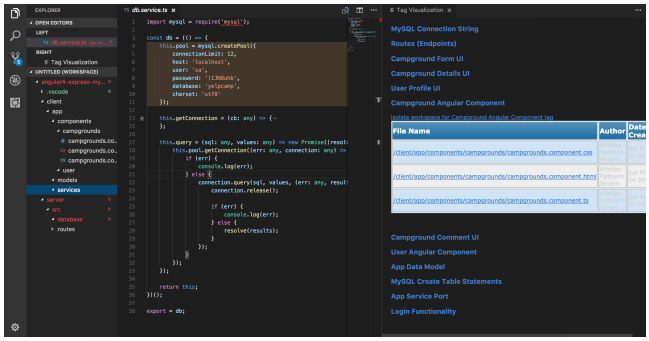

Figure 5: Isolation mode focuses the workspace in on a single Tag by folding methods and hiding files which do not contain an Instance of the Tag.

In normal development mode the tool enables the user to invoke a QuickPick menu to create a new Tag by supplying a name, or create an Instance of an existing Tag, as seen in Figure 1. We decided to use the metadata portion of Instances to store the author, Instance creation date, which file it belongs to, and its range. The range of an Instance is the line number on which it starts and ends, and is how Pontem associates an Instance with a code segment.

In addition to normal development mode the other part of our tool is isolation mode, which can be seen in Figure 5. Isolation is enabled when the user decides to highlight a Tag, as seen in Figure 2. Through the use of code folding and file hiding, isolation mode

focuses a user's workspace environment in on a given Tag. An example of file hiding can be seen in Figure 3. Isolation mode can either be entered through the use of a key binding or by using our Dashboard as seen in Figure 4.

The Dashboard can be used to view the list of defined Tags, as well as the source code files that contain the Instances of each Tag. It is integrated as part of the developer's workspace in the form of a panel that can be opened by invoking a command or keyboard shortcut. The panel displays an accordion list of Tags that can be used to navigate through a codebase based on Tags and Instances. Furthermore it is possible to quickly jump to a code segment associated with an Instance by clicking on the file name. This enables the user to quickly view all the different parts of a codebase which belongs to a user defined Tag.

Tags and Instances are stored as JSON files which gives plug-and-play support for version control tools such as Git, Svn and Mercurial. This enables team collaboration without having to set up additional resources excluding the hopefully pre-existing version control. Furthermore it enables the user or other developers to create tools which utilizes Pontem's dataset.

5.2 Limitations

Our decision to write the tool as an extension to Visual Studio Code put some restrictions on what and how we could implement things. For example, it was impossible to fade out files in the source code directory and we opted to hide files instead. Code folding is also done using the built-in lexical analysis which gives great support for TypeScript and JavaScript but is lacklustre when it comes to other languages. The current version of Visual Studio Code is lacking in HTML support which made aggregation and modification of Tags for our Dashboard difficult, which resulted in a mostly bare bones Dashboard. More webview functionality is on the way in a future release [15] and a possible version 2.0 of Pontem would support a richer Dashboard.

Currently all creation of Tags and Instances is the sole responsibility of the user. Having some smart logic such as static analysis of the code to be tagged or dependency tracing could help bootstrap the extension. This could be done through suggesting that a MySQL/PostgreSQL/MongoDB dependency facilitates a "DB" Tag, and code that uses an external dependency such as the aforementioned ones should with high probability be tagged with an according Tag.

It is difficult to tag an existing codebase with the current implementation of the tool since every Tag and Instance has to be done manually one by one. Having a mode for quickly tagging an existing codebase might make adoption easier.

As mentioned above the location of an Instance is on line numbers which gives the user great control over how to use the tool but might induce option paralysis by having the user question when and where to tag. Line numbers is very logical when it comes to defining a code segment but creates a sub-optimal abstraction between Instances and the code they belong to. Deciding another scope (function/class/file) might change how a user uses the tool and research into scoping of an Instance would be of great value.

6 EVALUATION

To evaluate our code tagging model we performed qualitative interactive demos and interview sessions with four participant developers. Each session took approximately 1 hour and we recorded the participants responses in real-time during the interview.

We sought to answer three primary research questions:

RQ1 Are tags helpful to new developers when they are first exposed to an existing codebase?
RQ2 Can a tagging tool help reduce clutter in a programmer's development environment?
RQ3 Can tags improve team awareness?

6.1 Setup

To prepare for the demos we obtained a codebase that was representative of a medium size program with enough lines of code and complexity to ensure that any task we asked the participants to perform was not trivial. The codebase used was a full stack web application built from Angular4, Primeng, Express, MySQL, Gulp and Passport that simulated a Yelp type site for campground reviews [7]. We then went through the application and tagged multiple sections of code as features and other functionality that the application contained. This included such things as the applications Login functionality that spanned multiple code blocks across five different files, its API endpoints for each of its database calls, and the multiple files involved in its UI that added new campgrounds to the database.

6.2 Interactive Demo

With a standardized tagged codebase we performed interactive demos following the script that we had defined in Step 1 of our evaluation interview template that can be found in Appendix A. In short it involved us carefully demonstrating the full functionality of Pontem: Navigating across files and instances, isolating the Visual Studio Code environment on a given tag and creating tags and instances.

6.3 Navigation Tasks

We had the participants perform two sets of navigation tasks before asking them open ended questions about their experience and thoughts on using Pontem for the defined task. First was a set of tasks that had participants use Pontem for navigation and identifying section of code where a requested change would be made. This included where to edit the MySQL connection string information for the application, what code blocks to change if a modification to the profile page was needed, and what files were involved in the applications login functionality. The second set of tasks involved isolating the Visual Studio Code workspace for a given tag using the isolation mode of our tool.

6.4 Tagging Tasks

We then had participants perform a set of tagging tasks where they were to first create a specific campground endpoint tag within the endpoints feature of the application. Followed by freely tagging other sections of the application's codebase as they so desired.

6.5 Research Questions to Interview Questions

As shown in Table 1, we categorized our interview questions and their corresponding participant responses into multiple topics. From this table we attempted to draw conclusions that either support or refute our research questions.

7 RESULTS

The results from our interviews are presented in Table 1. The topics and questions discussed are divided into four categories: general, tag navigation, tag isolation, and tag creation. The responses of the participants were generalized and listed alongside each topic.

8 DISCUSSION

The results shown in Table 1 give us some insight into how developers perceive our code tagging model. Feedback was generally positive, with most participants agreeing that the tool would be useful in practice. While the low number of participants makes it difficult to generalize our findings, we can nonetheless draw some useful conclusions to apply to our research questions.

In addressing RQ1, we discuss whether tags would be helpful to new developers when first exposed to a new codebase. From our results in Table 1, we are unable to answer this question directly from participant responses. However, by looking at the task completion rate during the interview sessions, we see that all participants successfully completed the navigation and isolation tasks. From this we can infer that on an unfamiliar codebase with the help of a tagging model, a new developer would find the code tagging useful. That being said, it is important to keep in mind that our low number of participants and the fact that our study design did not directly test this reduces our confidence in this claim.

Participant responses suggest that a tagging tool can help reduce clutter in a programmer's development environment (RQ2). From the interview responses our participants felt that, when compared to code-level comments, tags offer a more "streamlined" approach and could "reduce clutter". Furthermore, the participants responded positively to the tool's isolation feature, with three of the four participants stating that they like how it only shows relevant files. And one stating that they wished they could "write code in isolation mode".

To address RQ3, we initially had planned to conduct a longer, team-oriented survey in order to evaluate team awareness using our tool. However, we were unable to acquire access to an undergraduate student group on which to conduct a study. We therefore did not design our interview questions to address this research question and are unable to evaluate it in our work.

In general, the feedback from our participants was positive. Most negative responses voiced by the participants were related to the tools implementation, and not the model itself, which does not reduce our confidence in the usefulness of our code tagging model. Most of the negative responses also were less about disliking the current implementations or features, but about missing features and possible additions to the tool that they believed would be useful.

There were a number of responses that we found interesting and worth mentioning. First, one participant was noticeably skeptical about using tagging and saw no reason to switch to tags from code-level comments as it did not fit their coding style. Another response of interest suggested that entire lines should not be highlighted, and that the highlighting should be done in a less obtrusive way. Some of the participants also suggested additional fields to add to the Instance metadata, such as Instance description, line numbers for the block, and other tags that exist in that block. Also, three of the four participants expressed interest in the idea of "nested tags", which was not included in our implementation of the tool. Lastly one participant responded that they wished to be able to write code in isolation mode. We hadn't considered this possibility as it was meant to be purely a visual tool, but we agree that the ability to edit code while focusing the workspace on a Tag could be very useful.

9 THREATS TO VALIDITY

Construct Validity

The subjects were asked to answer knowledge questions regarding a codebase they have never seen before, which was tagged by one of the authors. This is prone to both *researcher expectations* and *defining predicted outcome too narrowly*. With such a small codebase it is hard to both show the power of tagging with having multiple Tags as well as not making the knowledge questions obvious. Furthermore our idea is susceptible to the same problem as documentation, more precisely, the fact that "anything is better than nothing". This makes it hard to separate if Tags are helping the user find something enough to justify the cognitive effort necessary to tag code. We tried to combat this last point by having the user tag some code themselves and then asking if it felt as a natural part of their workflow.

Statistical Conclusion Validity

The evaluation has low statistical power due to low sample size. This makes any conclusions drawn from the data dubious at best.

Internal Validity

The subjects were picked by asking friends and colleagues. This is prone to selection bias since we might have picked people that are more likely to give us the answers we want. As well, the subjects might also have a harder time being honest in their responses. Another possible threat for our evaluation is *confounding* which can happen if the subject is knowledgeable about the codebase we tagged, or has previous experience with the frameworks used in the codebase. Such a subject would therefore be able to quickly find the data we asked for without using the information avenue we were testing (tagging). This was combat by asking the subjects about their experience with Angular as well as if they had seen the project before. However, previous experience might affect their ability without being obvious enough to mention in the questionnaire.

External Validity

Our evaluation was only done against undergraduate and graduate computer science students at The University of British Columbia. How well the results from the evaluation translates into the real-world among more seasoned developers is unknown.

Ecological Validity

The evaluation was done as a one-on-one test with one of the authors and sometimes at their home which is not a typical work setting for a developer. To combat this we initially wanted to perform a study on an undergraduate team over a longer period of time which would test both the collaboration as well as how people

Table 1: Overview of survey results

Topic/Question	Response	Subjects
General		
Tag highlighting	Liked the non-intrusive look	2
	Can be distracting	2
	Entire line should not be highlighted	1
	Default state should be more neutral	1
	Not very aesthetically pleasing	1
When compared to code-level comments	More streamlined	2
	Tags might be more useful for new developers on a team	1
	Contains more metadata than code-level comments	1
	Code-level comments are more stable with group work	1
	See no reason to switch from code-level comments to tags	1
Incorporating external documentation	Too far from tagging to be intuitive	1
	Currently possible with comments, might be more useful to use with tags instead to reduce clutter	1
	Can be integrated with metadata	1
Tag navigation		
Grouping code blocks under a specific tag	Useful feature	2
	Should do it by files and consider the scope	1
	Tags don't need to be so granular	2
Metadata attached to the tags	Useful feature	2
	Similar to information you can already get from git	1
	Should add a field for description	1
	Should add a field for the lines of the block	1
	Should add a field for other tags that exist in the block	1
In what situations would you use this feature?	Joining a project late in the development process	1
	Using as documentation for group/team projects	2
	Would not use, does not fit coding style	1
What did you like about this component?	More flexible annotation than what is possible with comments	2
	Ease of use	2
	Good alternative to searching by comments	3
What did not you like about this component?	Nothing	1
	No support for nested tags	3
	Cannot delete tags	1
Tag isolation		
In what situations would you use this feature?	Working in group/team projects	2
	Any coding situation	2
	Debugging	1
What did you like about this component?	Collapsing of files and functions	1
	Only shows relevant files	3
	Visual aid is very helpful	1
What did not you like about this component?	Nothing	1
	No ability to write code while in isolation mode	2
What other information would you like to see in this component?	Ability to isolate multiple tags simultaneously	1
	Date the code was last modified within a tag	1
	Ability to sort and search	1
Tag creation		
Could you see yourself incorporating tag creation into your work-flow?	Yes	3
	If a team decides to use it, then yes	1
What did you like about this component?	Context menu and shortcut key work well	1
	QuickPick interface is non-intrusive	1
	Smooth experience	2
	Better and more usable than commenting	2
What did not you like about this component?	Nothing	3
	Auto-tagging and suggestions would be useful	1

use Pontem in a real-world setting. However, we were unable to do so due to time constraints and project complications.

10 FUTURE WORK

This paper has done a preliminary study into the idea of tagging code with some promising results. Future work into reducing the limitations expressed in Section 5.2 would be of great value. We only briefly implemented collaboration and were unable to test this as a part of our evaluation. Having a more team oriented approach to code tagging might reveal some flaws or strengths with the idea. Furthermore this is a field that cries for data aggregation and data visualization - it is trivial to imagine a dashboard which can be used for managers to quickly observe the cohesion and coupling of a codebase based on tags. In the current version of Pontem it is not possible to add external resources to Instances. However, such a feature were expressed among the participants as something of great value.

Some participants also suggested having a nested approach to tasks. For example being able to have a general "DB" tag and then sub tags such as "Connection", "Query", etc. This is not supported by our model but extending the model is possible.

This paper only tested the idea of tagging code without exploring how to best do it. There are multiple papers [4, 16] which focuses on how to create a user interface and how to visualize the relationship between different data structures. Approaching the idea from an UX perspective might give further insight.

11 CONCLUSION

In this paper we introduce a model to increase developer program comprehension via code tagging. Our tool Pontem implements the core ideas and components of the model. An evaluation using qualitative interactive demos and interview sessions was conducted to determine the usefulness of our model.

The results from our evaluation are promising and show that such a tool might help developers hit a solid middle ground between code-level comments and external documentation. However, this is a very new field and this paper is far from conclusive. Proponents of our model interested in pursuing future work should refer to Sections 5.2 and 10 for ideas on how to further extend our work.

REFERENCES

[1] Andrew Begel and Beth Simon. 2008. Struggles of New College Graduates in Their First Software Development Job. *SIGCSE Bull.* 40, 1 (March 2008), 226–230. https://doi.org/10.1145/1352322.1352218

[2] P. Boutet, K. Jensen, and L. Palazzi. 2018. CodeTagging. https://github.com/HoboKristian/CodeTagging. (2018).

[3] Microsoft Corporation. 2018. Visual Studio Code. https://code.visualstudio.com. (2018). Accessed: 2018-03-24.

[4] MEHDI DASTANI. 2002. The Role of Visual Perception in Data Visualization. *Journal of Visual Languages & Computing* 13, 6 (2002), 601 – 622. https://doi.org/10.1006/jvlc.2002.0235

[5] J. Espinosa, Sandra Slaughter, Robert Kraut, and James Herbsleb. 2007. Team Knowledge and Coordination in Geographically Distributed Software Development. *J. Manage. Inf. Syst.* 24, 1 (July 2007), 135–169. https://doi.org/10.2753/MIS0742-1222240104

[6] Carl Gutwin, Reagan Penner, and Kevin Schneider. 2004. Group Awareness in Distributed Software Development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW '04)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1031607.1031621

[7] Laurence Ho. 2017. angular4-express-mysql. https://github.com/bluegray1015/angular4-express-mysql. (2017).

[8] Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 344–353. https://doi.org/10.1109/ICSE.2007.45

[9] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[10] T. C. Lethbridge, J. Singer, and A. Forward. 2003. How software engineers use documentation: the state of the practice. *IEEE Software* 20, 6 (Nov 2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

[11] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 31 (Sept. 2014), 37 pages. https://doi.org/10.1145/2622669

[12] André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. 2014. Software Developers' Perceptions of Productivity. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 19–29. https://doi.org/10.1145/2635868.2635892

[13] Roberto Minelli, Andrea Mocci and, and Michele Lanza. 2015. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 25–35. http://dl.acm.org/citation.cfm?id=2820282.2820289

[14] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi. 2014. Quantifying Program Comprehension with Interaction Data. In *2014 14th International Conference on Quality Software*. IEEE Press, Piscataway, NJ, USA, 276–285. https://doi.org/10.1109/QSIC.2014.11

[15] mjbvz. 2018. Webview Api Prototype #42690. https://github.com/Microsoft/vscode/pull/42690. (2018). Accessed: 2018-03-25.

[16] Emerson Murphy-Hill and Gail C. Murphy. 2014. *Recommendation Delivery*. Springer Berlin Heidelberg, Berlin, Heidelberg, 223–242. https://doi.org/10.1007/978-3-642-45135-5_9

[17] Margaret-Anne Storey. 2006. Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future. *Software Quality Journal* 14, 3 (Sept. 2006), 187–208. https://doi.org/10.1007/s11219-006-9216-4

[18] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. 2006. Shared Waypoints and Social Tagging to Support Collaboration in Software Development. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW '06)*. ACM, New York, NY, USA, 195–198. https://doi.org/10.1145/1180875.1180906

[19] M. A. Storey, J. Ryall, J. Singer, D. Myers, L. T. Cheng, and M. Muller. 2009. How Software Developers Use Tagging to Support Reminding and Refinding. *IEEE Transactions on Software Engineering* 35, 4 (July 2009), 470–483. https://doi.org/10.1109/TSE.2009.15

[20] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2017. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2734091

APPENDIX A EVALUATION FORMAT

A.1 Overview of Tool

(1) Introduction to the codebase
   (a) https://github.com/bluegray1015/angular4-express-mysql
   (b) Briefly show them the running web app that the code base corresponds to.
(2) Open the project
   (a) Open the project "ANGULAR4-EXPRESS-MYSQL" expand all subfolders in "client" and "server" directories
   (b) Remember to copy over the .html file for tag-viz
(3) Run them through the tagging tool
   (a) To have a visual of the available tags press Ctrl/Cmd+Shift+Z or in the command pallet type "Show Tags".
   (b) Take them through the accordion panes that represent each individual tag.
   (c) Inside those accordion panes is a table with each instance of that tag (each row in the table) it has the file the tag is in, the author and date it was made on.

(d) Clicking the file link will open the tagged file in the adjacent text editor panel and reveal that location of the tag.
(e) Whenever a user clicks a tag isolation link it will collapse all foldable code and hide all directory tree elements that don't contain the tag.
(f) If a user clicks another tag isolation link it will change the UI to isolate for the newly clicked tag.
(g) To undo isolation at any point press Ctrl/Cmd+Shift+R.
(h) Command line options for tag isolation: pressing Ctrl/Cmd+Shift+R will bring up the isolation menu where you can select a tag to isolate the environment on.
(i) Pressing Ctrl/Cmd+Shift+R again will reset the environment.
(j) To create new tags start by selecting some code or having the cursor on the line you want to tag.
(k) Pressing Ctrl/Cmd+Shift+T to bring up the menu.
(l) On this menu you can select an existing tag to tag the code as or select "Create New" which will prompt you for a tag name. Pressing enter will create the tag and tag the code as that tag.

A.2 Task: Using the Tool for Navigation

(1) Procedure
(a) Have interview participants do the following navigating and exploring tags tasks.
(b) Browse through the tag visualization feature and jump to different instances of tags.
(c) Find where to edit the connection string to the MySQL database. (single file test)
(d) If you needed to output a different variable to the profile page what files would you edit? (2 file test)
(e) If you had to find out what developer to ask about the Login Functionality could you?(3 file test)
(2) Follow-up questions
(a) Do you find this tag hierarchy and navigation useful? How come?
(b) In what situations could you see yourself using this feature?
(c) Is grouping code blocks under a given tag useful?
(d) Was the meta information useful? (filepath, author, date)
(e) If you had to speak to someone about the tagged feature could you on the given information?
(f) Is there anything you did or did not like about this feature?
(g) Do you think tagging individual lines are best or would you like to see some other scope?

A.3 Task: Isolating a Workspace for a Given Tag

(1) Procedure
(a) Have the participant, using either the shortcut Ctrl/Cmd+Shift+R or the visualization UI, isolate the environment on a given tag.
(2) Follow-up questions
(a) Do you find this kind of dynamic context switching feature useful? How come?
(b) In what situations could you see yourself using this feature?
(c) Is there anything you did or did not like about this feature?
(d) What additional information would you want in such a view of the environment?

A.4 Task: Using the Tool to Create Tags and Instances

(1) Procedure
(a) Find the file that contains routes for campground.
(b) Highlight one of the endpoint functions.
(c) Create a new tag called "campground endpoint" and apply that tag to the code.
(d) Let the participant create other tags freely.
(2) Follow-up questions
(a) Could you see yourself incorporating this process into your coding workflow?
(b) Is there anything you did or did not like about this feature?

A.5 Other General Questions

(1) What did you think of the way which tags were highlight in the code? (colors, etc..)
(2) If we incorporated external documentation about the feature into the tags, say links to wiki pages regarding the tag, would this be useful?
(3) How would you compare it to code-level comments?