CodeStory: Capturing Rationale for Program Comprehension

Felix Grund and Nick Bradley
Department of Computer Science
University of British Colombia
Vancouver, BC, Canada
ataraxie,nbrad11@cs.ubc.ca

ABSTRACT

Software quality is strongly dependent on developers' knowledge of the rationale that lead to code changes. Capturing this rationale would significantly improve program understanding tasks leading to higher software quality. Unfortunately, this is a hard problem because decisions are often made unconsciously and are not being tracked. This is partly due to the fact that there is little tooling support and partly due to developers and managers putting little value in this information.

In this paper we describe a simple method for capturing developer's reasoning during coding tasks. Our implementation, called CodeStory, integrates transparently into a developer's workflow by hooking into the standard copy-paste operation. When developers copy text or code snippets from StackOverflow, CodeStory will capture additional context and include a reference to it with the pasted content in their code. This information can later be seen by other developers giving them a better understanding of the original developer's rationale.

1. INTRODUCTION

Software is the result of decisions made based on information obtained from many sources. While software implicitly captures the results of these decisions, their rationale evaporates unless some additonal process is used to capture it. In practice, documenting knowlegde is often considered a resource-intensive process without tangible, short-term gains, so often it is skipped or performed inadaquately [1, 2]. However, we found that it is exactly this rationale that helps developers provide effective code reviews and could help with other comprehension tasks.

Developers do not program in isolation: they use resources on the internet and colloborate with other people to accomplish a programming task. Communication tools such as email, instant messaging and web sites like StackOverflow make it easy for our tool to capture knowledge as it is transferred between individuals. This is due to the fact that these forms of communication are already in a text-based representation and, more importantly, follow a typically concise format of question-discussion-answer. We suggest that by linking these knowledge transfers to the sections of code that prompted the inquiry, significant value can be added in the form of increased future comprehension.

To exploit this, we created a tool called CodeStory that captures the rationale behind this question-discussion-answer communication and links it to the resuling code. It does this by inserting a comment with a hyperlink pointing to a web page that describes when and where the information came from and what the original question was; i.e. the story behind the code.

CodeStory currently supports the popular developer Q&A site StackOverflow as the knowledge transfer medium. On this site, developers post technical questions about a problem they have encountered and then, after reviewing several possible solutions contributed by other developers, they choose the one that best fits their requirements. We take the action of a developer copying some or all of the content of an answer, typically a code snippet, as indicating their choice. At this point we collect additional information that forms the code story for the copied snippet and include a link to it when the snippet is pasted into code.

The contributions of this paper are as follows:

- A simple approach for capturing text-based collaborative rationale among developers as it is transfered from one developer to another in question-discussion-answer type interactions. This can include communications using email, instant messaging, and websites. The approach is also appropriate when the developer has a question that can be answered by consulting digital documentation.

- A prototype tool that implements this approach. It captures contextual information surrounding snippets copied from StackOverflow using a Google Chrome extension and a server-side database. An Atom[1] package lets developers paste the snippet with a hyperlink pointing to a web page showing the corresponding code story.

2. RELATED WORK

Several papers have focused on documenting design rationale in a disciplined and structured manner. Kyaw, et al. [3], for example, describes a multi-dimensional design space tool called CoDEEDS that allows development teams to share knowledge by recording design decisions and rationale. Instead of linking to the codebase, their system links to artifacts. They define artifacts as any things (or work products) produced as a result of design and development. It is designed to explicitly capture high-level meta-information such as modelling information, design constraints, and process information in a centralized repository accessible through CoDEEDS. The meta-information is organized to allow traceability of design decisions as they are made. Bratthall, et

[1]https://atom.io

al. [4] followed a similar approach of requiring developers to explicitly record design rationale using a separate system. They evaluate if the upfront cost of these systems reduces future costs by speeding up changes and improving correctness.

Our work differs in several ways. First, we avoid the need of a separate system and directly integrate with existing developer workflows to reduce the number of different processes present in the development cycle. Second, we link the rationale directly to snippets in the codebase instead of focusing on larger artifacts. This results in the links being distributed throughout the codebase instead of in a centralized system (although the code stories are stored in a database). Finally, CodeStory is designed to capture information as developers ask questions; other tools are focused in supporting formal design discussions. We followed a similar evaluation methodology as that used by Bratthal but we focused on evaluating program comprehension through code reviews instead of evaluating performance on programming tasks.

3. APPROACH

Anything that does not directly result in usable code is an extra cost to development companies. However, capturing decision rationale is important for future development. Thus, we want to minimize the upfront cost of the capture process while simultaneously maximizing the future benefits of this captured information. Taking these constraints into consideration we decided on four key design decision criteria for CodeStory.

Minimize impact to source code. Developers do not want to be distracted by unnecessary or verbose comments in their code. Because our tool hooks into the copy-paste operation, it can capture a great deal of information but including all of it as comments in the code is not desirable. After trying several different levels of comment verbosity, we concluded that a single-line comment would be the most appropriate even though it would require extra steps to see the full code story. Extending our tool to support displaying the story in the IDE would help mitigate this. Not including any comment would be ideal but that presents many challenges for linking the story with its corresponding code.

Transparently integrate into existing workflows. A tool must be used if it is to provide any benefit. Since CodeStory captures information useful to future developers, it is imperative that it not require any extra effort on the part of the current developer. Hooking into the standard copy-paste operation was an obvious way to capture information that the user was interested in bringing into their code without requiring any explicit actions.

Our tool consists of three components, each of which is easily modifiable, making it easy to extend CodeStory to support user's existing tool sets. CodeStory automatically captures contextual information when a copy command is invoked in a supported source program and pasting with the code story hyperlink can be done by a user defined keybinding, including the default paste keybinding. The code story is stored separately from the code: the insertion of a standard one-line comment is the only modification to the code.

Capture the right contextual information. The goal of CodeStory is to capture and describe the rationale behind decisions made in code without requiring further action from the developer. We had to decide what information would be most pertient to this task: capturing too little would have made it hard to understand the rationale while too much would have obfuscated the key points. To help us decide, we used a pilot survey which we describe in section 5.1.

Present the captured information in a meaningful way. To allow developers to benefit from a code story during program comprehension tasks it is important that the story be close to the target code, viewable in any tool, and be easy to interpret. As mentioned before, CodeStory inserts a hyperlink directly above the pasted code. By default, the code story can be served as an HTML web page so that it can be viewed in most IDEs and in any web browser. It can also be served in other formats such as JSON and XML to make it viewable in custom applications. The code story summarizes the raw captured information in natural language making it easier to read and understand.

4. IMPLEMENTATION

CodeStory is implemented as three components: (1) a Chrome extension responsible for collecting contextual information surrounding text copied from StackOverflow, (2) a backend receiving this information and persisting it in a database, and (3) an Atom package that extends pasted snippets with a link to a page showing this information. Here we briefly describe the implementation details of each component and how they interact with each other.

4.1 Chrome Extension

The CodeStory Chrome extension mainly consists of a Content Script that is executed whenever a user visits a StackOverflow page. An event listener for the browser's *copy event* is added and called whenever content from the page is copied to the clipboard.

The listener first collects the required contextual information from the current page and saves it in a *storyData* object whose fields are shown in Table 1. This object is `POST`ed to the backend with a unique ID (the first ten digits of the MD5 hash of the timestamp and stringified object) whenever a user copies content from StackOverflow. We ensure that the backend accepts cross-origin requests so that the Chrome extension can simply use `Ajax` calls.

In order for our Atom package to be able to link to the contextual information, we include the ID in the clipboard content using the JavaScript clipboard API[2]. The Atom package then uses the ID to form a URL which is included in the pasted content.

4.2 Backend

Our backend consists of a web server implemented in Node.js using the restify package and a redis database. The database stores the *storyData* object using the ID mentioned in the previous section as the key. The web server provides three endpoints for creating and viewing the *storyData* object. In particular, it provides two `REST` endpoints, `POST /codestory/rest` and `GET /codestory/rest/:id` for creating and retreiving the object, respectively, and a `GET /` endpoint for serving HTML files. An example of a CodeStory web page is shown in Figures 3 and 4, which can be requested by using the URL in the CodeStory comment (see Figure 2).

[2]https://developer.mozilla.org/en-US/docs/Web/API/ClipboardEvent

Field Name	Description
copiedFrom	Whether the snippet was copied from a question or an answer.
type	Whether the copied snipped is code-only or a combination of code and text.
originalSelection	Exact text that was selected in the browser upon copy.
questionTitle	Title of the question of the current page.
questionUrl	URL of the question (i.e. the URL of the current page).
questionContent	Full content of the question as text.
questionContentWithHtml	Full content of the question preserving the HTML markup.
questionVotes	Number of votes the question received.
answerUrl*	URL of the answer.
answerContent*	Full content of the answer as text.
answerContentWithHtml*	Full content of the answer preserving the HTML markup.
answerVotes*	Number of votes the answer received.
accepted*	Whether the answer was accepted.
accessTime	Timestamp when the page was accessed.
fullCodeSnippet	The full code snippet (only available if the selected code is part of a code snippet).

* Only available if copied from an answer.

Table 1: Fields collected from StackOverflow upon copy

4.3 Atom Package

Atom is a popular, multi-platform, text editor that supports extending functionality with packages written in JavaScript. We created a simple package to provide enhanced pasting: if the clipboard content came from our Chrome extension then it will be inserted into the Atom editor with a comment above which includes a URL to the corresponding CodeStory web page. Otherwise, the standard paste operation is performed (i.e no comment is inserted). The correct commenting characters are used when pasting content into any file type supported by Atom. For example, `// View code story:`*`URL`* would be inserted into a JavaScript file while `# View code story:`*`URL`* would be inserted into a Python file.

Maintaining expected paste behaviour was an important design choice that will affect developer adoption of the tool. We ensured that only clipboard content ending with the special tag, `CodeStory:`*`<HASH>`*, would alter the default paste behaviour. Further, we set the default key-binding to be CTRL+SHIFT+V but the user can change the binding to override the default paste command by setting the binding to CTRL+V. Alternatively, the paste operation can be invoked from the packages menu in Atom.

5. EVALUATION

Code reviews are an essential part for modern software engineering and greatly influence software quality. Therefore, we choose to evaluate CodeStory by having participants complete two code reviews each, one with comments from our tool and one without (section 5.2). We then analyzed the quality of the code reviews to determine if our tool had any effect in section 5.3. Before this study, however, we determined the general feasibility of CodeStory by performing a pilot survey among industry developers who perform code reviews on a daily basis (section 5.1). Threats to validity for our evaluation are described in section 5.4.

5.1 Pilot Survey

At an early stage of this paper, we aimed to determine whether our assumption that it is useful to record developers' reasoning during coding tasks is true. Additionally, we wanted to collect information on what information may in fact be useful. We therefore collected 17 responses of industry developers who regularly perform code reviews with a survey.

After providing a brief introduction of our research, we described the following scenario: *a developer has to choose a sort algorithm for a particular task. She googles "sort array in JavaScript" and finds a code snippet on StackOverflow. She copies the snippet as a scaffold into her code.*

We then let participants rate the following question on a scale of 1 (not useful) to 5 (very useful): *How useful would the story above be during the code review?* The responses to this question were positive: no participant rated with 0 and only 2 participants rated with 1 (negative responses). 6 participants rated with 3 (neutral), 7 participants rated with 4 and 2 participants rated with 5 (positive). Overall, 88% responded neutral or positive to the question whether recording the described contextual information would be useful.

Finally, we collected opinions on the particular information of interest using the same 1-to-5 scale: *What elements from the above story would you consider useful?* Table 2 shows the list of elements and the percentage of the responses that rated with either 3 (maybe useful), 4 (somewhat useful) or 5 (very useful).

Considering the majority of neutral and positive responses to our pilot survey we felt confirmed that the idea behind CodeStory is valuable and we could continue our research. The single element that had a overly negative result in the survey was the *Google search query leading to StackOverflow*. We consequently chose to not include this element in the implementation of CodeStory.

5.2 Study Description

In order to determine the usefulness of CodeStory in practice, we approached developers with code reviews. We created a bootstrap repository on Github that contained a simple Maven Java-Project. We aimed to simulate a realistic scenario where in a developer read a StackOverflow page, copied a code fragment and pasted it into their code. We chose two questions on StackOverflow as scenarios for the study:

Element	% Ratings ≥ 3
Google search query leading to SO	24%
SO question URL	76%
SO question heading	47%
SO question content	76%
SO answer URL	76%
Time of access of SO page	59%
SO answer code snippet	53%
Entire SO answer	47%
SO answer rating	59%
SO answer acceptance status	59%
SO answer comments	59%
Other SO answers	47%

SO = StackOverflow.

Table 2: Elements of interest for survey rating

1. *Gson - convert from Json to a typed ArrayList<T>*[3]
2. *How do you compare two version Strings in Java?*[4]

In (1), a developer was struggling with deserializing a JSON string to an instance of a custom Java class using the google-gson[5] library. The developer tries to pass a class object for the generic type `ArrayList<JsonLog>` by passing `ArrayList<JsonLog>.class` to `fromJson` (Gson's method for deserializing a JSON string), which is not valid Java code and does not compile (because class objects cannot be created from generics that way in Java). The first answer to the question, which is also the accepted answer, describes how to solve this problem by using Gson's `TypeToken` class for creating the desired class object: `new TypeToken<List<JsonLog>>().getType()`. Figure 1 shows the diff of the change that we created for this scenario.

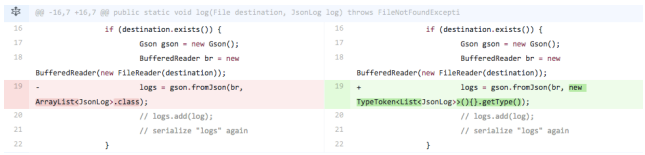

Figure 1: Diff view for study scenario 1

In (2), a developer was interested in how to compare version strings semantically. This question is associated with *semantic versioning*[6] which defines what versions should be considered higher than others, i.e. *1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-beta < 1.0.0-rc.1 < 1.0.0*. More specifically, the question was: *given two version strings a and b, which one is higher?* For this scenario, we chose a lower ranked answer[7] as a base version to be replaced by a higher ranked answer[8] as the improved version. Both are answers to the given question but neither is an accepted answer. We chose to use these answers because the accepted answer did not contain any code. Figure 2 shows the diff of the change that we created for this scenario (note that this diff shows a CodeStory annotation for illustration purposes).

[3]http://stackoverflow.com/questions/12384064
[4]http://stackoverflow.com/questions/198431
[5]https://github.com/google/gson
[6]http://semver.org/
[7]http://stackoverflow.com/a/27891752/1105907
[8]http://stackoverflow.com/a/6640972/1105907

Figure 2: Diff view for study scenario 2

For both scenarios we now added CodeStory to the developer workflow. The resulting CodeStory pages are shown in Figures 3 and 4.

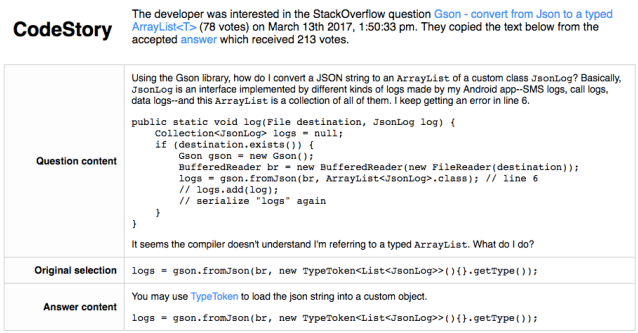

Figure 3: CodeStory page for scenario 1

For the changes described in scenarios 1 and 2, we created another version of each that included a link to the respective CodeStory page as URL in a comment above the pasted content. Since scenario 2 contained two paste operations (one in an XML file and one in a Java file), two such comments were shown in the diff. These two scenarios resulted in four treatments for our study:

- **T1:** Scenario 1 without annotation
- **T2:** Scenario 2 without annotation
- **T3:** Scenario 1 with annotation
- **T4:** Scenario 2 with annotation

The study was performed with 8 developers among which 4 participants were graduate students and 4 were industry developers (> 4 years of practical experience with Java). With participants from different backgrounds we ensured generalizability of our study to both academia and industry. Due to time constraints we decided to assign each participant 2 code reviews (we assumed approximately 10 minutes for each code review). To avoid learning bias for the given scenarios and treatments, we decided to let each participant do one code review for each scenario: one with annotation and one without annotation. To further avoid learning bias, we ensured that 4 participants start with a code review with annotation and the other 4 start with a code review without annotation. Table 3 shows the outline of treatments and participants.

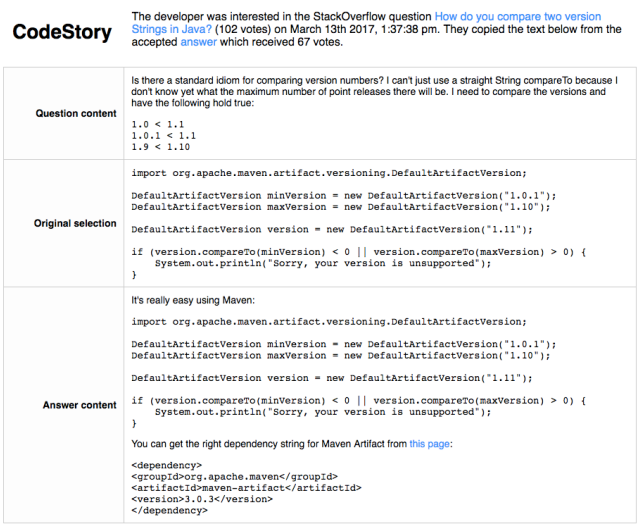

Figure 4: CodeStory page for scenario 2

Participant	Treatment	Background
P1	T3 + T2	industry
P2	T3 + T2	academia
P3	T1 + T4	industry
P4	T1 + T4	academia
P5	T4 + T1	industry
P6	T4 + T1	academia
P7	T2 + T3	industry
P8	T2 + T3	academia

Table 3: Study outline

For each participant we forked our bootstrap repository and created two pull requests with respect to their treatments. In the description of each pull request we asked the following questions:

- **Q1:** *What is the purpose of the method with the change?*
- **Q2:** *How did the method change?*
- **Q3:** *Why was this change made?*

We then added the participants as reviewers and instructed them to perform their code reviews in sequential order by answering our questions as comments on the pull requests. The code reviews were performed online without any further constraints.

In the instructions for the second pull request, participants were asked to complete a survey designed to determine the effectiveness of the CodeStory annotation. We first asked participants to rank their experience for each pull request on a scale from 1 (very challenging) to 5 (very easy). We then asked *Was the information provided by CodeStory useful?* Table 4 shows the results of the survey with the ratings of each participant and the associated treatment. Finally, we gave participants the option to comment on our tool: *Do you have any comments about CodeStory? Were you missing any information?*

We found two issues with our evaluation: (1) one participant (P4) did the pull requests in the opposite order and (2) another participant (P8) did not notice the CodeStory link

	Rating (1-5)				
Participant	T1	T2	T3	T4	Overall
P1	-	2	4	-	5
P2	-	2	4	-	5
P3	4	-	-	5	5
P4	1	-	-	5	4
P5	2	-	-	5	5
P6	3	-	-	5	5
P7	-	5	3	-	4
P8*	-	4	2	-	2

* Ignored in subsequent sections because participant did not notice the CodeStory tool.

Table 4: Survey results

in the diff. We consider (1) to be a minor issue that does not impact the results and (2) a major issue requiring us to *ignore the results for P8* in the subsequent analysis.

5.3 Impact for Code Review Tasks

Comparing code review comments without (T1/T2) and with CodeStory annotation (T3/T4), we could clearly see an increase in quality using our tool. Basically, T1 and T2 induce great amounts of guessing in finding the right answers to our questions, primarily visible through phrases like ”presumably”, ”it appears to be”, ”it seems to me”, ”I believe”, etc. A decrease in quality in answers to our questions Q1-Q3 is visible for T1/T2: while Q1 could often be answered relatively easily, Q2 proved very hard and Q3 proved nearly impossible. That is why we observe the most guessing with Q3. Some participants tried to do their own online research for the questions on T1/T2 but were mostly still unable to provide good answers. Others commented that Q3 is not answerable for T1/T2. While we could generally observe higher quality answers from industry developers than from graduate students, the relationship between code reviews without and with our tool remains the same and we can therefore neglect variance resulting from participants' backgrounds.

In contrast, comments on code reviews with CodeStory (T3/T4) showed significantly higher quality. Most participants provided very good answers to our questions for these treatments. The additional contextual information provided by CodeStory clearly helped them to answer what ”really happened” for the given code changed, i.e. what the developer's reasoning was during implementation. The following list provides representative examples of code review comments with answers to our questions for each treatment:

- **Scenario 1 without annotation (T1):** *Q1: I think that the method is trying to convert a json object into a gson object. Q2: It changed through the usage of the TypedToken List and anonymous class. Q3: The method is really small, though I believe that it was changed to consider the usage of interfaces and generics. As far as I remember, there is an issue trying to get the type of ArrayList<JsonLog>.class and the correct implementation would have to create an anonymous class.*
- **Scenario 1 with annotation (T3):** *Q1: The purpose of the method is to parse a JSON file to an ArrayList of the custom object JsonLog. In the original*

version, the parsing fails because of an unconventional definition of the target Java type. Q2: The method changed in the definition of the target type which is necessary due to Java's type safety. The type itself as well as the generic of this type are now passed on to the Json parser Gson indirectly, by first instantiating a dummy TokenType with a generic of the target type and, second, by calling .getType() on this dummy object. Q3: The change was made because java class objects cannot simply be passed on as parameters if they are modified by generics.

- **Scenario 2 without annotation (T2):** *Q1: Checks whether the two versions are the same. Q2: Instead of doing a character-by-character string comparison of the two versions, converts them to a DefaultArtifactVersion object, and then compares the objects. Q3: Hard to tell without knowing how DefaultArtifactVersion works (I could look it up but I don't want to); my guess is that the previous version is not robust to differently formatted strings that represent the same version.*

- **Scenario 2 with annotation (T4):** *Q1: The method compareVersions(a, b) determines the precedence of two version strings, a and b. Precedence is indicated by an integer return value: less-than-zero implies a < b, greater-than-zero implies a > b, and equal-to-zero implies a == b. Q2: The implementation was changed from a homebrew implementation to an off-the-shelf implementation using Maven. Instead of comparing the versions as strings, they are now parsed into properly typed objects implementing the Comparable<T> interface. Q3: (...) the programmer needed to implement a new feature or quirk to the version comparison logic that Maven already handles, such as comparing non-numerical version components (e.g. the "-rc1" in "1.10-rc1"), and didn't want to reinvent the wheel.*

Analyzing the results of the final survey further strengthens our positive impression of CodeStory. This is most clearly visible in the comparison of the code reviews for scenario 2 (recall that we neglected the results from one participant as described in the previous section). On our scale from 1 (very challenging) to 5 (very easy), participants rated the version without annotation (T2) with 2, 2, 5 whereas all participants rated the version with annotation with 5. Scenario 1 shows similar results, though not as visible as in scenario 2. In our interpretation, this is due to the fact that the answer on StackOverflow was a simple one-liner and that question as well as answer provided far less detailed information about the problem than in scenario 2. Participants rated the version of scenario 1 without our tool (T1) with 4, 1, 2, 3 whereas they rated the version with our tool with 4, 4, 3. In summary, *the average of the ratings without our tool is **2.71** while the average of the ratings with our tool is **4.43**.*

Analyzing the optional comments of the final survey, we noticed that one participant missed exactly the element of information that we neglected after our pilot survey, the *Google search query leading to StackOverflow* (see section 5.1): *The search term(s) which led the programmer to that version comparison StackOverflow question could have been useful for explaining why s/he made the change. Search terms such as "java version comparison algorithm", "java version comparison best practice", and "java version comparison pre-release metadata" might all lead to that question, but convey different intents/requirements.* Our interpretation is that the participants in the pilot survey might not have fully understood what we referred to with this element and they would most likely have considered it more useful in an actual code review.

5.4 Threats to Validity

While we tried to reduce the number of factors that could impact the validity of our findings there is a chance that our results may not be representative of the actual utility of CodeStory.

We evaluated the usefulness of the rationale captured by CodeStory for code review tasks. We requested participants view very short diffs in pull requests on a nearly empty bootstrap repository. This was intentional to keep the tasks short and focused but are not entirely representitive of real code reviews.

There is a potential for bias among participants and tasks. To mitigate this, we used four treatments and assigned two participants to each. One participant was from industry and the other was from academia to futher reduce bias. Additionally, to control bias among treatments, the first pull request in two of the treatments contained a CodeStory annotation while the other two had the annotation in the last pull request. In all treatments the diffs were different from each other to account for learning bias. Unfortunately, this meant that we could not directly account for variance among tasks but this effect should be reduced by assigning each treatment to two participants. Despite these controls and our positive results, it is possible that CodeStory would not be found favorable in general.

6. DISCUSSION

After describing our approach (section 3), implementation (section 4) and evaluation (section 5) of CodeStory, we now discuss our approach's potential and some of its drawbacks.

Potential. Our evaluation showed that CodeStory can significantly improve the *quality of code reviews*. While most of the participants of our pilot survey (section 5.1) were only neutral or slightly positive about the idea of our tool, participants of our main study (sections 5.2 and 5.3) gave us almost entirely positive feedback: our tool made their code review tasks significantly easier and they commonly rated it as very useful. By comparing the code review comments (qualitative analysis) and analyzing the results of the final survey (quantitative analysis) we could clearly see that participants had a *better understanding* of the code changes they were confronted with and could therefore perform the code review tasks much better. With our study scenario of code reviews as one task instance in the domain of program understanding, and given the assumption that high quality code reviews lead to better overall software quality, we can conclude that the concept behind CodeStory has the *potential of increasing software quality as a whole*. Besides its potential in terms of program understanding and software quality, we can imagine other benefits in tracking the described contextual information. For example, our dataset could be used for *analysis tasks inside a software company* to identify patterns among their developers and their decisions in order to improve overall development performance within

the company. Furthermore, multiple CodeStory datasets combined could become very powerful, providing valuable information about *developer behavior on a global level*. In particular, information on interaction patterns with StackOverflow and evaluation of copied contents in production code could be very useful for researchers, developers and managers.

Drawbacks. One negative aspect considering the current implementation of CodeStory is its *manipulation of the operating system's clipboard*. While we intended to design our tool to be as transparent as possible, we consider adding the ID of the associated CodeStory to the OS clipboard a major interference with the developer's environment. This issue could be mitigated by implementing a user management system that is shared throughout all components of CodeStory (Chrome extension, backend, Atom package). Furthermore, there may be a *performance impact* using our tool, since the tool requires access to the CodeStory server. This issue could be mitigated by hosting that server locally and minimizing the impact on connection problems (e.g. by simply doing nothing upon failure). While we consider the *footprint of CodeStory* minimal, one could still argue that sudden "comments out of nowhere" may be confusing for developers and may even clutter a codebase, especially when used inconsistently within a company. Countering this argument, we would reflect the benefits of using CodeStory against this issue. Another concern might be *social and privacy issues:* a feeling of "being tracked" might arise among developers using CodeStory. However, we are not claiming that our tool is appropriate for all environments and its benefits should always be contrasted with its drawbacks.

7. FUTURE WORK

The results of our study show that capturing and displaying contextual information makes code reviews eaiser. In order to make our study coherent and to keep it to a reasonable length for our participants, we decided to forego evaluating the CodeStory tool directly. However, there are several interesting questions that would help us improve CodeStory. Some of these include: Should more information be shown in comments? Does CodeStory fit into developers' existing workflows? Would developers keep the tool installed after a week of first using it?

We would also like to determine if the code story has different utility for different comprehension tasks. It would be very interesting to know, for example, whether bugs can be found and/or fixed more easily in annotated codebases. Or, if new features are incorporated into the software in a more consistent manner or in less time because developers have a chain of rationale to follow.

The prototype can be extended to support more sources like email clients and instant messaging applications which are common tools used by development teams. Currently, a user needs to open the code story in a web browser which can break their workflow. Extending our IDE package to support displaying the code story in the IDE would make it easier to take advantage of the code story. Adding support for other popular tools that are used for specific comprehension tasks is also important. For example, in the code review tasks given to the participants, we wanted to show the code story in a popup window in Github when the reviewer hovers over the link.

Finally, we would like to evaluate the effectiveness of our presentation of the code story.

8. CONCLUSION

In this paper we presented an approach for capturing decision rationale and provided a protoype tool, CodeStory, that implements this approach. Code implicitly captures the result of decisions but these decisions are often the result of question-discussion-answer type discussions. CodeStory enables developers to transparently collect this rationale by capturing contextual information surrounding copied text. We found that by making this information available to code reviewers, the quality of reviews was generally better. Currently, the prototype only works with StackOverflow but the approach applies more generally to other common communication applications like email and instant messaging. We have yet to see if this information applies to other tasks requiring program comprehension like bug fixes.

We developed our approach by conducting a pilot survey to establish what information on StackOverflow would provide the most utility to code reviewers. We then evaluated the usefulness of the captured rationale by having participants each perform two code reviews, one with a code story and one without. The participant's feedback showed that they found the code story helpful in performing the code review task.

Overall, we believe that capturing the rationale for question-discussion-answer type communications is important for improving software quality and that CodeStory provides a minimally distruptive way to do so.

9. REFERENCES

[1] O. Zimmermann, U. Zdun, T. Gschwind, and F. leymann, "Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method," in *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, WICSA '08, (Washington, DC, USA), pp. 157–166, IEEE Computer Society, 2008.

[2] N. B. Harrison, P. Avgeriou, and U. Zdun, "Using patterns to capture architectural decisions," *IEEE Softw.*, vol. 24, pp. 38–45, July 2007.

[3] P. Kyaw, C. Boldyreff, and S. Rank, "A design recording framework to facilitate knowledge sharing in collaborative software engineering," in *Chu, W*, (ACTA Press pp 70-88), November 2003. Citation: Boldyreff, C. et al (2009) ?A Design Recording Framework to Facilitate Knowledge Sharing in Collaborative Software Engineering? In: Chu, W. (ed) 2nd IASTED International Conference on Information and Knowledge Sharing, 17-19 November 2003, Scottsdale, Arizona, USA. ACTA Press pp 70-88.

[4] L. Bratthall, E. Johansson, and B. Regnell, *Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution*, pp. 126–139. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.