2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing

CodeTalk—Conversations About Code

Bastian Steinert*, Marcel Taeumel†, Jens Lincke*, Tobias Pape†, and Robert Hirschfeld*
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
http://www.hpi.uni-potsdam.de/swa
*Email: {firstname.lastname}@hpi.uni-potsdam.de
†Email: {firstname.lastname}@student.hpi.uni-potsdam.de

***Abstract*—Contemporary development environments do not directly and explicitly support developers in having a conversation about the code they write and maintain. This problem is aggravated when geographically dispersed teams need to collaborate on development artifacts. CodeTalk allows developers to have conversations about source code elements. They can mark code sections they are concerned about and annotate them. These annotations provide entry points for an informal discourse about the strengths and weaknesses of these sections and developers can work towards a conclusion on how to proceed on the raised issues. A Squeak/Smalltalk implementation of CodeTalk was evaluated by several small development teams, indicating improvement in the informal assessment of code.**

I. Introduction

The purpose of computer programs is the automation of tasks by telling the computer what to do and how; the development of these programs mainly involves communication amongst developers. Writing and extending programs can be considered as communication. Developers express parts of their understanding of a program's domain in a programming language of their choice, other developers read it. Reading source code and comprehending its intent is an activity developers spend much time on; it is the prerequisite for modifying a program—extending it with new functionality or refactoring it to a simpler design.

Programs can be written in different styles making them more or less easy to understand [1]. This leads to another kind of communication amongst developers having the source code itself as the topic. Programming guidelines such as [2] help teams to establish a common coding standard easing program comprehension. Pair programming, as suggested in [3], further encourages developers to continuously talk about the source code. It helps to ensure a good understanding of the program and helps developers learning to write programs that are easier to understand and maintain. Another form of communication about source code are comments, which have always been part of computer programs. Comments can be used to describe the intent of source code in a language different than the programming language. However, much care and discipline is required to keep comments and the source code in sync. For this and other reasons, it is generally desirable to write programs so that they can be understood without comments. To achieve this goal, developers have to communicate about the source code. All the sins and all the pieces that smell bad have to be brought to light, allowing developers to learn from them and to improve the program.

Communication about source code in software development teams is of significant importance, but still it is not well supported by current development environments. Tools encouraging this kind of communication are even more required for distributed teams, as geographical dispersion impedes collaboration. Traditional approaches to communicating about code such as comments or emails are inadequate; comments as well as emails are not explicitly connected to source code entities of interest, requiring significant effort to describe this connection. This unconnectedness makes comments and emails also likely to be ignored or forgotten.

In this paper, we suggest CodeTalk as an approach to enable developers to talk about source code in an informal manner. CodeTalk has been implemented in Squeak Smalltalk [4] and was inspired by reviewing tools for digital documents such as PDF. In a similar way, CodeTalk allows for marking selected source code of interest and adding annotations. In addition, the markup is exchanged along with the source code using source code management systems. With that, communicating about issues, such as non-meaningful variable names or a need to refactor, becomes easy and practical, requiring less effort and providing a better separation of concerns.

The remainder of this paper is organized as follows: Section II describes the target scenarios of our work and points out the need for effective means to informally communicate about source code related issues. In Section III, we present our approach to support development teams in this respect and describe the underlying concepts. Selected aspects of our implementation in Squeak Smalltalk are discussed in Section IV. We describe the conducted case study and our results in Section V. Section VI discusses related work and Section VII concludes.

II. Background and Motivation

In this section, we motivate the need for tool support that enables efficient informal communication about source code

We gratefully acknowledge the financial support of the Hasso Plattner Design Thinking Research Program for our project "Agile Software Development in Virtual Collaboration Environments".

978-0-7695-4029-0/10 $26.00 © 2010 IEEE
DOI 10.1109/C5.2010.11

related issues. We emphasize the importance of these concerns in teams working geographically dispersed, and describe the limitations of traditional ways to communicate about source code.

A. Communicating About Code

Developers often talk about the source code of the system to be developed and extended. The source code itself is the most important artifact during the development process, particularly in agile development processes. Developers usually care deeply about it and prefer, for example, simple and elegant solutions over complex ones that are more difficult to understand and maintain [3]. The system naturally evolves and is extended; so developers often have the need to talk about the source code with others.

Software developers spend much time reading code. Using, extending, or modifying parts of a system's source code requires an in-depth understanding, ranging from the intended use of interfaces to the interplay of multiple, independent system parts. However, parts of the system may be difficult to comprehend raising the need to request support from the originators; an algorithm might be very complex or the intended run-time behavior might be difficult to infer [5].

During code reading developers also often discover source code that needs to be revisited and improved; for example, variable or method names can be too general and thus not very meaningful [1]. Developers might further have ideas to simplify the system's design [3], [6] or even detect potential failures in algorithms. While these issues are often discovered during regular coding activities, developers may not have enough time or background knowledge [7] to refactor the respective parts of the system or to validate their theory of a failure and fix it if necessary. And sometimes, developers would rather like to continue working on their primary task at hand [8]. So, an efficient mechanism is needed to make the discovered issues explicit and share their insight with peers.

The need for communication about code also arises during the creation and modification of code entities. Developers might explicitly want to ask other developers for help or remarks, for example, whether the defined interface matches the expectation or whether there is a better, simpler solution. So, communicating about source code in general is an important aspect of software development, that should be supported well.

B. Collaboration in Geographically Dispersed Teams

The need for communication support concerning source code related issues increases when software development teams are geographically dispersed. Working together in one office encourages informal talks about the program domain, the source code, and other topics that are of interest. Developers can program in pairs, as suggested by Extreme Programming, and can discuss about the source code at hand whenever necessary. Developers further have a good awareness of other developers' activities; regular meetings, coffee breaks and lunch, and other joint activities support the exchange of knowledge. In co-located settings, additional communication takes place not involving active speaking or listening; developers passively hear the discussions of others. So, developers usually know what their colleagues are working on right now, the tasks they worked on, the problems they tackled; and developers are aware of their colleagues' habits concerning work in general and programming style in particular. This knowledge and opportunity for informal discussion supports the communication about source code related issues such as non-meaningful variable names or the need to refactor. There is, however, the trend that project teams tend to disperse around the world. Distributed development is getting more common, requiring team members to resort to means other than face-to-face communication to organize themselves, to collaborate, and to keep in touch regardless of geographical location. Developers have fewer opportunities for informal talks and are less aware of each others' activities and habits. The barriers to communication are much higher, involving the potential of omitting discussions about the rather small source code related issues. Tools that developers currently use to exchange ideas and knowledge are not satisfactory regarding the desired informal ad-hoc kind of communication.

C. Traditional Ways of Communicating about Source Code

Current approaches to communicating about source code include *source code comments* and external communication tools and protocols such as email and instant messaging. However, both have limitations and do not encourage conversations about code.

Comments allow developers to document the source code. They can also be used for reviewing and adding remarks or questions but have the following restrictions and limitations.

- The *belonging* to code entities referenced in the comment is often ambiguous. Making the belonging clearer, developers often insert additional blank lines, indicating whether the lines below or above are referenced in the comment.
- Additional description is required when authors want to reference a particular element in the code, such as a particular variable name
- Comments further get out of sync with the referenced code easily, also due to the informal character of the connection between comments and referenced source code.
- Developers hesitate to modify source code invasively for adding an informal comment spontaneously; they might respect the ownership, a carefully designed code layout, or they might just have a different mind-set while reading the source code similar to reading physical paper documents.
- Only adding a comment is not sufficient to request the attention of other developers. In a sense, it is just one more comment.

The issue mentioned last, the need to point other developers to concerns and entities of interest, is usually handled by sending an email. Developers might also copy the source code of interest, paste it into an email, and describe the remarks or questions. Writing emails, however, has the following

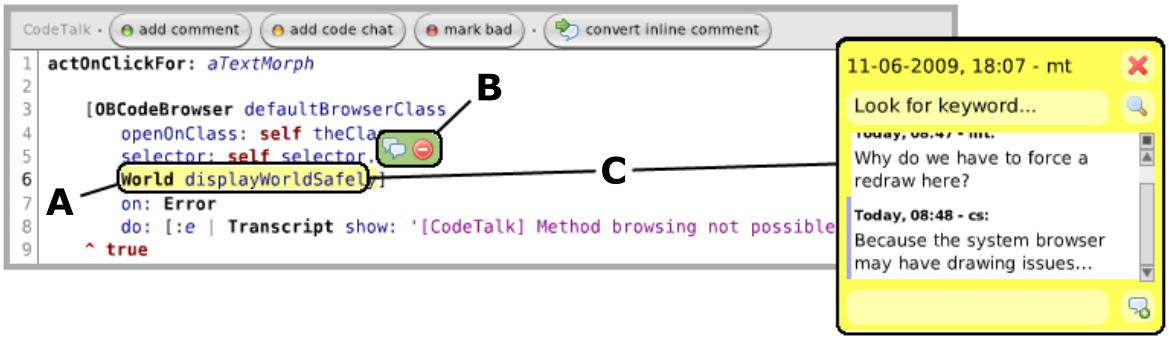

Figure 1. Code with markup (A), markup halo buttons (B) and the chat (C).

additional disadvantages, mainly due to email clients not being integrated into the development environment:

- It involves a context switch and requires a considerable amount of time, contradicting the idea of informal communication and ad-hoc annotations. In the focus of a primary task at hand, writing an email might be considered as too much effort to notify others about inadequate names.
- It requires detailed descriptions of the code entities referenced in the explanation, as emails are not connected to the source code.
- Issues described in emails are easy to forget, when they are not handled immediately.

Neither comments, nor emails, nor their combination provide adequate means to support informal spontaneous communication about source code. But this kind of communication is important; it helps to ensure a high code quality and helps developers to become better in their profession. This has been our motivation to design and develop a new approach to support this informal ad-hoc communication that we describe in the next section.

III. CodeTalk—Informal Communication via Markups

This section describes CodeTalk, our approach to enable efficient informal communication about source code. CodeTalk, which was implemented in Squeak Smalltalk [4], allows developers to mark and annotate single expressions, whole lines, or entire methods in the source code. It works similarly to text processing applications and tools that are capable of adding comments to PDF files. The markup and annotations are shared along with the source code using regular source code management support.

A. Annotating Source Code

Developers regularly come across pieces of source code that are of unsatisfying quality and bring up the need for discussion. Not all kind of issues can, however, be addressed immediately, as we pointed out above II. By means of CodeTalk, developers can mark the the source code of interest and describe their concerns. This annotation mechanism enables developers to capture even small issues and bring them to light. Team member will become and stay aware of all issues.

One team member might, for example, discover an invocation of an expensive operation. Figure 1 shows an example method in a typical code browser in Squeak. The annotated statement forces a full redraw of the entire scene graph, which can be a time-consuming operation. Developers might be skeptical about the necessity and mark the selected code as critical using a context menu or a keyboard shortcut. This will highlight the statement with a red background color. To additionally describe their opinion and thoughts, developers add a note in the dialog that will be displayed next to the marked section, as shown in Figure 1.

This new annotation functionality was integrated into the standard development environment, in particular into the tools for browsing and editing the code. So, developers can informally annotate a piece of code whenever necessary during their regular coding activities. The region of interest in the source code can be marked directly and annotated with an explanation.

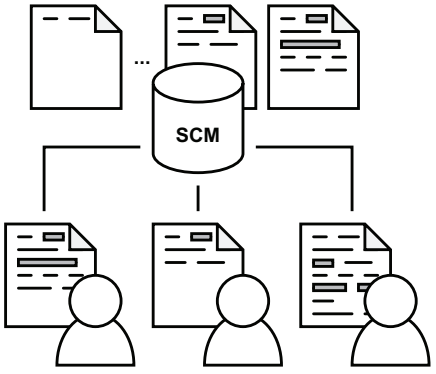

Figure 2. CodeTalk's markups are shared through the SCM.

B. Exchanging Annotations

Annotations are an integral part of the source code and as such they are exchanged along with the source code itself. When developers commit modifications applied to their working copy, they will also submit all annotations currently

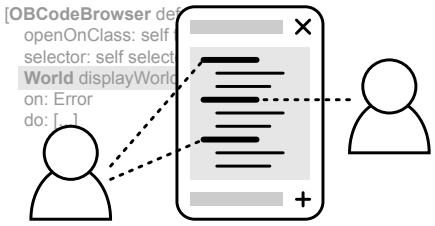

Figure 3. A new conversation about code evolves.

in the code base to the source code repository, as depicted in Figure 2. The critical question about the statement that force a complete redraw is now part of the newly created source code revision.

When team members update their working copy later, they will retrieve the newly added annotation along with source code modifications. After the regular code update procedure a new dialog window will appear providing information about new or changed annotations, shown in Figure 4.

Other developers will then notice the question regarding the redraw statement, and the authors of that code might either remember a reason for forcing the redraw or they might not. In the later case, they might consider removing the statement, test the application to validate the assumption, and commit the modification. As the annotations are connected to the source code they reference, the annotations would be removed together with the referenced statement in the described scenario.

On the other hand, if forcing the redraw is required, developers can change the type of annotation, from critical to normal, and answer the previous question. Our extensions to the code browser enable developers to reply directly to questions or remarks in annotations so that a chat can evolve (Figure 3).

CodeTalk was designed so that every annotation refers to a specific revision of the source code. All annotations are thus persistent. This additionally allows for browsing older annotations of a method that were already removed.

C. Browsing Markups

The markup browser of CodeTalk as seen in Figure 4 appears after each code update to provide an overview of all annotations. It draws the developers attention to new issues and allows developers to deal systematically with the marked issues, so that they are not forgotten or overseen in the editor.

This alternative view provides easy access to all markups appearing in the source code and addresses the awareness problem by filtering (A), grouping (B) and sorting all markups of a specific scope like a package or a class. The browser presents general information like creation time, creator, a hyperlink to the method and the respective code snippet (C) and lists all chat messages in a reversed chronological order (D) below.

D. Hyperlinking in Annotations

Talking about source code often involves other sources located outside the currently discussed context: Sometimes

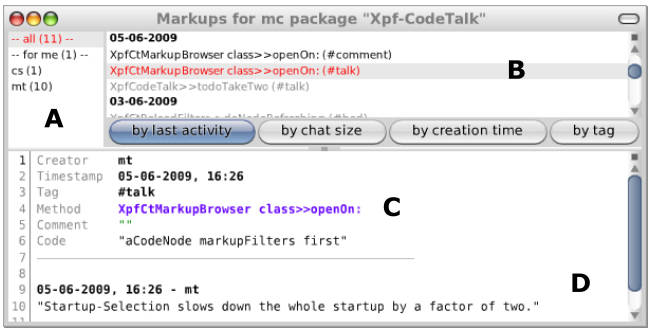

Figure 4. The markup browser increases the awareness level of new annotations.

developers come across methods that seem to be very similar, but they do not have the time or knowledge to perform the necessary refactoring. CodeTalk allows developers to mark that issue and to reference the other method in their comment. For example, the chat in Figure 5 replaces the occurrence of "String»#findTokens:" automatically with a hyperlink that browses to the method "findTokens" in the class "String". The link below points to a method "split" that does not exist and is therefore drawn in red.

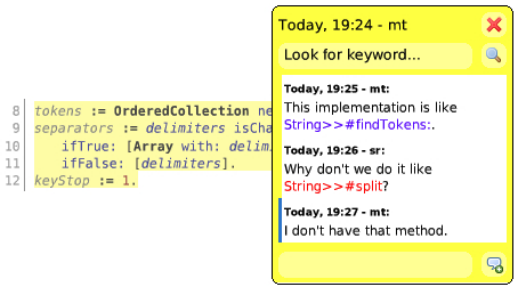

Figure 5. Hyperlinks from comments to methods enable convenient source code navigation.

E. Discussion

CodeTalk provides informal semantic classification of its markup using background colors. Normal syntax highlighting does not aim for classifying the highlighted source code in any way. However, the ability of CodeTalk to mark a code snippet red, yellow, or green provides classification. Generally, the resulting background color is independent from the programming elements. Developers can consider this in many different ways. Thus, it is possible to assign different meanings to colors, e. g., to point out critical sections or bad style.

The ease of use encourages the developer to prefer a markup to a classic inline comment. Besides using the context menu, there are keyboard shortcuts and a toolbar above the editing panel present to address different types of users. Additionally, the colors that are available for markup form an explicit form of highlighting that is easily recognized. Thus, the variety of tools

combined with the eye-catching markup provide advantages over using comments and are more likely to be used.

The primary concept of CodeTalk is to separate the discussions about the source code from the source code itself, while keeping them connected. This separation allows for individual support for the different concerns; specially designed tools can ease the creation and exchange of annotations and can provide a better awareness of these issues. The direct connection between source code and annotations indicates that they belong together and, thus, encourages developers to keep both in sync. This may prevent the problem that occurs when the code gets updated while accidentally ignoring the corresponding comment.

IV. Implementation of CodeTalk

This section describes selected aspects of CodeTalk's implementation in Squeak Smalltalk. Our implementation heavily benefits from how text is represented in Squeak. CodeTalk is also integrated with a source code management system that enables, for example, the distribution of annotations along with the source code.

A. Source Code as Rich Text

The implementation of CodeTalk is based on the ability of the Squeak environment to handle source code as rich text. The Squeak environment also provides text processing capabilities to work with text objects and formatting these texts. Due to these capabilities, Squeak has been prepared to handle source code as rich text as well. This feature originally enabled developers to individually style source code, for example, to color segments of special importance or to change their font size. While this feature has been ignored by recent developments such as the automatic syntax highlighter[1] or the source code management system Monticello, our CodeTalk implementation makes use of these capabilities.

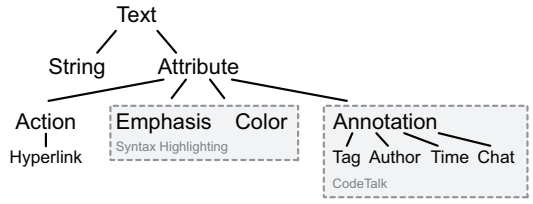

Figure 6. Relationship between text and annotations

Regarding Squeak, every text object has attached markup information describing the desired style and format of the text. The markup technique is based on a concept called *RunArrays* that is similar to standoff markup but does not feature overlaps. Every style (like bold or italic) is represented by attribute objects and every region of a text with a different set of attributes is a single run. These attributes are then used during the actual rendering of the text. We introduced markup attributes that are drawn as colored rectangles during the text rendering. The position of the marked text, which is calculated in the rendering process, is then used to update the markup halo buttons. These halo buttons are the user-interface elements for adapting the region, deleting the markup, and adding or showing an annotation to that markup.

B. Comments and Talks as Annotations

CodeTalk's annotation text attribute is associated with tags and textual comments. The textual comments can be talks, which are chats of different authors. The annotation preserves time stamps and authors, to allow filtered views in the markup browser.

The automatically created hyperlinks in text are created by parsing the text with regular expressions and replacing everything of the format "aClass»#methodName" with a proper hyperlink.

Comments and talks are displayed in their own windows, to separate them from the display of the source code.

C. Source Code Management Integration

We extended Monticello,[2] which is the standard source code management system (SCM) in Squeak, to support markups (see Figure 2). CodeTalk associates annotations with text objects that are then handled by the source code management system. Traditionally, this SCM only operates on strings of source code. However, it was possible to make Monticello aware of text object-based source code and make it store annotations in packages.

V. Case Study

CodeTalk has been used by several developers during a case study that was performed with 80 students in our *Software Engineering I* lecture. Students formed 16 different teams of five that were asked to develop applications in Squeak. The teams used an agile software development process such as *Extreme Programming* [9]. The project's time frame was about three months.

After the project, the source code of all revisions of all groups was analyzed for markups. As the agile process that had to be employed by the teams incorporated primarily co-located work, there was no necessity for asynchronous communication; most of the students met every day due to lectures to be attended. However, not all students were able to work like this. Long travel times, work responsibilities, or child care hindered co-located work. Thus, we found that four teams made strong use of CodeTalk annotations that have been stored using the Monticello SCM. In order to gather personal experiences with CodeTalk two of the four teams that were using CodeTalk were interviewed.

A. Interview

During the interview several questions were presented to the teams. In the following, the essence of the teams answers is presented

[1]Project website: http://www.squeaksource.com/shout/ (2009-10-22)

[2]Project website: `http://www.wiresong.ca/Monticello` (2009-10-19)

		Team 1	Team 2	Team 3	Team 4
Number of Methods		745	605	700	828
Number of Revisions		178	174	246	335
Number of Markups	All Over Time	103	25	82	43
	Maximum	50	9	22	17
	At Project End	0	1	2	3
Average Lifetime		33	18	27	63

Figure 7. Summary of markup usage from selected teams

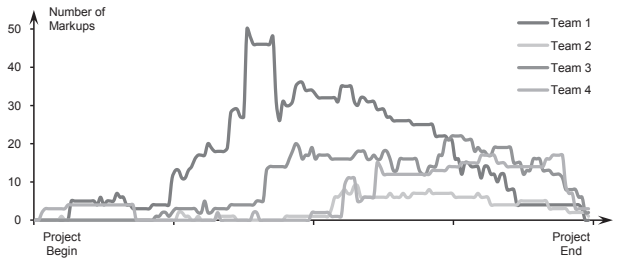

Figure 8. Absolute number of markups in the source code over whole project development time

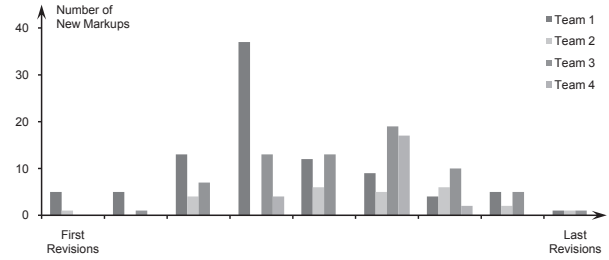

Figure 9. Number of new markups in several development parts

a) Application of CodeTalk: The teams reported that markups were used to write down tasks. This included planned refactorings of bad source code and new features that needed to be implemented. Additionally, the *critical* markup was occasionally used to point out bad coding style. Students also used comment markups for personal notes, especially as ToDo-items.

b) Reasons of usage instead of, e. g., email: Those teams that made heavy use of CodeTalk actually had a strong need for asynchronous communication, as many team members contributed from many different location and at different times for several reasons. The students argued that they used CodeTalk mainly due to convenience; it allows for staying in the current environment and context instead of switching tools.

c) Reaction to Markups: While CodeTalk offers the possibility to address an annotation to a particular developer, this feature was not used in most cases and team members usually wanted to address the whole team. Also, developers were always aware of annotations, understanding them as part of the code base. The presence of markup had such a strong effect that developers were actually concerned about an increasing number of markups that they might not be able to handle.

d) General impression of CodeTalk: The markup browser was judged to be clearly arranged and helpful.

e) Problems: It was stated that it would have been helpful to add annotations to describe a new feature. Currently, this feature is not present, for annotations are associated with source code. Often, not yet implemented features do not have any corresponding source code, thus, no annotations are possible.

B. Data Analysis

We additionally evaluated the actual use of CodeTalk during the projects by analyzing all source code revisions. As shown in Figure 7, the analyzed projects are of similar size consisting of about 600 to 800 methods. While one team created 20 annotations, other teams created up to 100 annotations.

Figures 8 and 9 indicate a continuous use of CodeTalk during the course of the project. At the end of the projects, development teams cleaned up all markups, hopefully handling the described issues first. Note that the source was inspected by teachers at the end of the project. The average lifetime of annotations was 20 to 60 revisions, approximately a fifth of all revisions created during the project time.

Annotations created during the projects include the following examples:

- "That is somehow totally crap. The instance variable *separatorMap* seemed to be good for defining the place of these separating things for each category ..." (from German: "Das ist irgendwie total Mist. Die Instanzvariable *separatorMap* dacht ich wär gut, um für jede Rubrik festzulegen, wie die Trenndinger stehen müssen ...")
- "Looks paradoxical! ..." (from German: "Irgendwie paradox! ...")
- "Where should the layout code be included, this seems not to be a good place?" (from German: "Wo soll das Layout stehen? Hier ist vielleicht nicht der beste Platz.")
- "onClick + callback => nonsense"
- "Yes, there is a better way to do this :-)"

C. Discussion

Developers started to use CodeTalk occasionally in the beginning of the projects and used it more often later on (Figure 9). It seems that the need for conversations increases with the size of the code base. We further think that developers regard annotations as part of the source code and understand critical annotations as an indicator for insufficient code quality. All remaining annotations were related to the end of a project, to make it ready for release.

The example annotations listed above show that developers like to use a colloquial style for communicating about source code related issues. While we have no evidence whether the teams would have discussed a similar amount of issues without CodeTalk or not, we think CodeTalk actually encourages this

kind of conversation, which is important to bring all flaws to light.

VI. Related Work

The markup notion of CodeTalk has similarities to the review markup concepts of common document processors. Regarding the programming domain, CodeTalk is not the first attempt to deal with developer communication by means of source code. The notion of Literate Programming shares some rationale with CodeTalk. Also, two tools are known to have similar aims as CodeTalk.

A. Review Markup

Document processing applications often support adding review markup to their documents [10]. As an example, consider PDF annotation supported by Adobe Acrobat or the free PDFedit[3] and Xournal[4]. These application provide graphical and textual annotations that are stored inside a PDF file in order to serve as review markup. Similarly, text processing applications feature review markup; e. g., OpenOffice.org[5] provides a review toolbar that facilitates markup insertion and tracking of changes to content. Both kinds of application focus on direct communication between the reviewers, which is similar to CodeTalk's intent. However, these tools are not applicable in software development, as the scope of their document formats does not include source code.

Regarding tool support for writing in dispersed teams, an overview of such tools that support annotations is available in [11].

B. Literate Programming

Introduced by Knuth, Literate Programming [12] (LP) facilitates logically structured documentation by extracting program code and documentation into separate entities from a common source. That source, in turn, is logically structured as *nodes*. Nodes and, thus, pieces of code (e. g., functions) that are used in different places of the source code are referred to by cross references in the documentation, thus providing high code navigability. Kasper Østerbye has introduced an LP environment for Smalltalk [13] that features a hypertext-approach for the aforementioned cross-referencing facility. Both Østerbye's and Knuth's approach are verbose and encourage incorporating rationale about written code. They also include source code-invasive techniques such as splitting methods across documentation nodes when they are actually one method in the target source code. Furthermore, Østerbye's approach features hyperlinking of all words to possible targets nodes in the environment, especially classes and methods. This is similar to the hyperlink mechanism of CodeTalk.

LP focuses on documentation and less on addressing other developers or asking questions. High code readability is also an aim of LP.

C. Source Code Markup Tools

Two tools are known to be intended to handle markup in source code.

a) TagSEA: Extending the *Javadoc*[6] tool, TagSEA [14], [15] tries to combine *waypoints* and *social tagging* in comments. Tags written according to the Javadoc conventions are processed and presented in separate windows to allow easy navigation from tag to tag. Whole routes can be created through the code. A case study [15] proves that the use of tags and informal messages can produce an information catalog which helps to understand and develop a system. These *stand-in* comments are accessible with every notepad application and, due to Javadoc, not limited to Java as programming language. However, an intensive use of Javadoc comments could make it hard to read the source code itself. CodeTalk is able to exploit the possibilities of Rich Text, thus, does not need to affect comments. Although the structure of markups in CodeTalk is quite flat compared to TagSEA, limited possibilities regarding custom tags result in a collection of markups that is much easier to handle by all developers.

b) ICICLE*:* Intended as a code inspection tool to be used in code review processes, ICICLE [16] has been developed to add annotations to source code. During review meetings, this tool facilitates the recording of remarks regarding the source code. Resulting *stand-off* markups needed a tool to visualize them together with the source code. ICICLE supports the association of an annotation to a line of source code. Small icons at the beginning of each line indicated their presence. The purpose was to optimize the system, discover bugs and discuss other concerns in the context of a review session. Using CodeTalk, direct communication and asking questions is possible during the entire process of code writing.

VII. Summary and Outlook

In this paper, we suggest CodeTalk as an approach to support informal communication about source code. Adequate means to support this conversation are required to encourage the discussion about issues such as non-meaningful variable names, the need to refactor, performance considerations, or suspicion of a bug. The demand for effective communication support is increased in geographically dispersed development teams. CodeTalk, which was implemented in Squeak Smalltalk, addressed these needs by enabling developers to mark and annotate a selected piece source code, and by automatically exchanging these annotations along with the source code. Our approach to annotating selected source code is based on handling it as rich text in the development environment. CodeTalk was successfully used in student projects. During these projects, students added several annotations to their code base, bringing inadequate pieces of code to light. Our plans for future work include the integration with other development tools such as issue tracking systems or test and integration tools, which continuously report about the health of the entire

[3]Project website: `pdfedit.petricek.net` (2009-10-22)
[4]Project website: `xournal.sourceforge.net` (2009-10-22)
[5]Project website: `www.openoffice.org` (2009-10-22)
[6]Project website: `http://java.sun.com/j2se/javadoc` (2009-06-27)

system. We also plan extending support for discussions about further software entities and system structure. This will enable developers, for example, to easily describe the need to refactor a class hierarchy.

VIII. Acknowledgements

We thank Ian Piumuarta for his valuable feedback and fruitful discussions.

References

[1] P. Oman and C. Cook, "Typographic style is more than cosmetic," *Commun. ACM*, vol. 33, no. 5, pp. 506–520, 1990.

[2] E. Klimas, S. Skublics, and D. Thomas, *Smalltalk with style*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995.

[3] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley Longman, 2004.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," in *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 1997, pp. 318–326.

[5] A. Von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[6] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[7] T. Shaft and I. Vessey, "The relevance of application domain knowledge: Characterizing the computer program comprehension process," *Journal of Management Information Systems*, vol. 15, no. 1, p. 78, 1998.

[8] E. Horvitz, C. Kadie, T. Paek, and D. Hovel, "Models of attention in computing and communication: from principles to applications," *Commun. ACM*, vol. 46, no. 3, pp. 52–59, 2003.

[9] K. Beck, *Extreme Programming Explained: Embrace Change. ISBN 0201616416*. Addison-Wesley, 1999.

[10] J. Wolfe, "Annotation technologies: A software and research review," *Computers and Composition*, vol. 19, no. 4, pp. 471 – 497, 2002.

[11] R. M. Baecker, D. Nastos, I. R. Posner, and K. L. Mawby, "The user-centered iterative design of collaborative writing software," in *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*. New York, NY, USA: ACM, 1993, pp. 399–405.

[12] D. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, p. 97, 1984.

[13] K. Østerbye, "Literate smalltalk programming using hypertext," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 138–145, 1995.

[14] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2006, pp. 195–198.

[15] M. Storey, L. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall, "How Programmers can Turn Comments into Waypoints for Code Navigation," in *IEEE International Conference on Software Maintenance, 2007. ICSM 2007*, 2007, pp. 265–274.

[16] L. Brothers, V. Sembugamoorthy, and M. Muller, "ICICLE: groupware for code inspection," in *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*. New York, NY, USA: ACM, 1990, pp. 169–181.