CodeTalk: Source Code-Related Communication in Distributed Agile Teams

Marcel Taeumel

Bachelor's Thesis, June 2009
marcel.taeumel@student.hpi.uni-potsdam.de

Abstract. The success of a project developed by a distributed team depends on effective communication. As agile software development processes claim to react fast to changing requirements, these conversational topics often address the application's implementation. CodeTalk provides a way to annotate source code with chats. Unlike classic inline comments, these transparent markups colorize a text's background to avoid distraction on account of informal messages. This project was realized in *Squeak*, an implementation of *Smalltalk*. A case study reveals the usability of such a tool as well as the needs in this situation that will be satisfied in contrast to e-mails.

1 Introduction

Agile software development methodologies are prepared to react shortly to changing requirements. *Extreme Programming* [1], as one popular representative, depends on effective communication between all developers. *Pair Programming*, combined with shifting pairs, and *Collective Code Ownership* ensure a consistent understanding of the internal application structure. Additionally, a short conversation during lunch with the whole team can dispel any existing misconceptions. An important assumption is, that work schedules of all developers overlap for the most part and therefore one team member may reach another in person to discuss a topic.

Nowadays, big companies with branch offices all over the world are able to carry out projects with distributed teams. Several communication issues arise when regarding distances and different time zones. In the last resort, no common working hours can be organized between the developers.

This bachelor thesis analyzes CodeTalk, an attempt to realize efficient communication about source code in Squeak. After finishing this section with a motivational example, section 2 lists all main features derived from their requirements and needs. In section 3, the implementation details are shown. Starting with a basic explanation of code with styling information,

all necessary integration points in the given environment are mentioned. Evaluation details of a small case study is covered in section 4. Section 5 provides remarks to related work. Finally, section 6 summarizes all results and some ideas in regard to future work.

1.1 Motivational Example

A hypothetical team of four developers is working on a project at two different locations: Palo Alto and Berlin. So the time difference is nine hours—shifting of almost a whole working day is possible. The team members are called Dave, Joan, Alex, and Lucy. All are experienced programmers with a lot of team spirit. The following two situations will illustrate some problems with effective communication in this setting.

Situation I: Reading New Code. It is Tuesday morning in Palo Alto. Dave and Joan meet each other at work and check the repository for new source code. They notice that the Europe sub-team wrote a new component some hours ago and both try to read the code. In spite of their knowledge, they have difficulties understanding it. Unfortunately, a telephone call is meaningless because the office of Alex and Lucy is not staffed anymore. Joan opens her instant messaging client to chat with Alex but his online status says: "Not available."—no answer. This is problematic because their current task is affected. Moreover, Dave discovers an inline comment which addresses himself but he can't figure out its intent. The need of efficient asynchronous communication about code arises.

Joan could write an e-mail but then she would have to copy the source code or describe its position in the project. Probably an answer would appear in her inbox some hours later and more mails would be necessary until the problem is solved. Having such a long discussion using e-mails, it could happen that the connection to the source code gets lost. Dave could add another inline comment to answer Alex or Lucy but such an informal dialog besides productive code lines distracts other reviewers too much. There is a chance, that the addressed person would never notice the comment. In the end, working time would be wasted.

Situation II: Writing Code with Comments. It is a normal working day in Berlin. Alex and Lucy are programming together. Lucy is the *driver* [1] who writes the code and Alex supports this with observation and suggestions. They are working on a task that addresses login problems with new application users. Finally, the bug is fixed but they are not happy

with the solution. They want to tell the other team members why this code is so unsatisfying. Communication with Palo Alto is not possible due to the time shift.

Lucy could defer a check-in, but that wouldn't be very agile because the bug is fixed and so the task completed. Another developer, who has an idea how to optimize the code, could refactor the solution later. *Continuous Integration* [1] encourages the progress of the whole project. Version control systems like SVN[1] allow to add a comment for each check-in but just like writing an e-mail, a connection to the respective source code needs to be created. Assuming that not every new source code line is affected, some lines need to be copied or their location needs to be described again. Some developers may even never read those comments when checking out more than one new version. Using bug tracking systems, a custom state could indicate that a ticket is "fixed but bad". In the end, a defined place is needed, where the developer can write down its concerns for the solution.

1.2 Vision of Efficient Asynchronous Communication

The developer wants to add annotations to any scope or area of source code. This includes single words, lines, whole methods, classes or other units according to the particular programming language. Annotations can contain a wide range of information, like a chat, an image as well as audio and video clips. Everything has to be visualized behind the code—not above or below. There should be no distraction from the source itself. Any additional tool that fulfills this request should integrate seamlessly into the development environment of the developer's choice. The used version control system should take care of these annotations and offer awareness of their updates to other developers. That's the way, the creator can be sure to address the person or group he intended to.

2 Code Annotations in Squeak

This section covers a high-level view on CodeTalk. Starting with the requirements of all development stages, all relevant features are going to be explained. Concurrently, their integration into the workflow of Dave, Joan, Alex and Lucy from the given setting will be described. The terms *markup* and *annotation* as well as their equivalent verb forms will be used synonymously.

[1] Project website: `http://subversion.tigris.org` (2009-06-27)

2.1 Changing Requirements

CodeTalk was developed using *Extreme Programming* [1]. Therefore all requirements evolved during the implementation. A customer created and defined the priority of all feature wishes and was able to discard one or another.

Non-Functional Requirements. The Squeak[2] Smalltalk environment has to be extended to support source code markups. Along with that environment, *Monticello* [2] should be considered to manage various versions of annotations and distribute them to all developers. To build on top of a useful and extensible tool chain, all features have to work in the package browser of the *OmniBrowser* [3] framework. Everything has to support *Shout* which is responsible for syntax highlighting. This should provide a convenient base because most developers are familiar with source code that is colorized and easier to distinguish.

Functional Requirements. All customer requests can be divided into three steps. The first one creates a basic architecture. Some selected source code should get a background color—green, red or gray. Semantically, this should indicate good or bad code as well as a neutral marking. Color information has to be persistent for a method after saving its source and visible after displaying that method again. Adding a background color should be possible with a keyboard shortcut, the context menu of the selection and later some kind of toolbar in the package browser. There are several *views* for a source code possible in the code browser. The *source* view should show the code styled by Shout and therefore stay as it is. Additionally, a new *CodeTalk* view should show code with syntax highlighting and the background color if present. That *markup*, how the background color is called, should be versioned with Monticello. It is important, that a developer who does not have CodeTalk installed is able to check-out from the repository and read the code. No other visible meta information should distract from the code then.

In a second step, the background color should evolve into a container for more information. A code chat should be possible. Chat lines should store the author's initials, the current system time and text message to enable a talk between all developers about a specific code snippet. The meaning of background colors is nonsatisfying for the customer. Green is not needed anymore because code without annotations was meant to

[2] Project website: `http://www.squeak.org` (2009-06-27)

be good code. Only neutral chats and chats about bad code have to be distinguished. Markups with neutral chats have to be changed from gray to yellow to be more like a note. Small inline morphs [4] beside a selected markup should open the chat or delete the markup on a click. This is intended to reduce mouse movement when interacting with code annotations. Small icons after the method name in the list of methods should indicate a markup somewhere in the source code.

Finally, supporting the awareness of markups becomes most important. A special browser should show them and categorize them by their last activity. All information from the markup ought to be presented clearly arranged. Easy navigation from that markup summary to the method with the markup in a normal code browser has to be possible. The markup browser should open after each check-out with Monticello. Every chat message should offer a hyperlink when recognizing the pattern `class>>#selector` to support easy navigation between several places in the code. There ought to be a local history to notice markups that were already read. Old markups have to be grayed-out in the markup browser. Green markups should be back to allow inline comments in the form of a free text field.

2.2 Talking About Code

CodeTalk offers three different ways to add an annotation to the source code of a method. The fastest one is a keyboard shortcut (*CMD+T*) that tags the current selection with a code chat. Alternatively, using the context menu or the integrated toolbar may be more intuitive for the user.

Figure 1 shows the typical environment when working with markups: The affected source code gets a background color (A) and small buttons near the annotated text afford straightforward editing facilities (B). One of those buttons opens the chat behind the code (C). New messages have to be inserted in C2 and are listed in C1 after clicking the button in the bottom right corner (C3). Markup and chat window have the same coloration. Code marked as *bad* has a light red background and a code chat in the same tone. There is a seamless integration into a Squeak programmer's workflow. No specific user interface is needed to make markups persistent or transport them with Monticello.

In situation I, the team in Palo Alto could just add a chat to the incomprehensible piece of code. They could ask questions and give an opinion about the implementation. There would be no distraction from the code itself because another developer would only see the chat if he wants to. Even classic Smalltalk comments could still exist, they can get

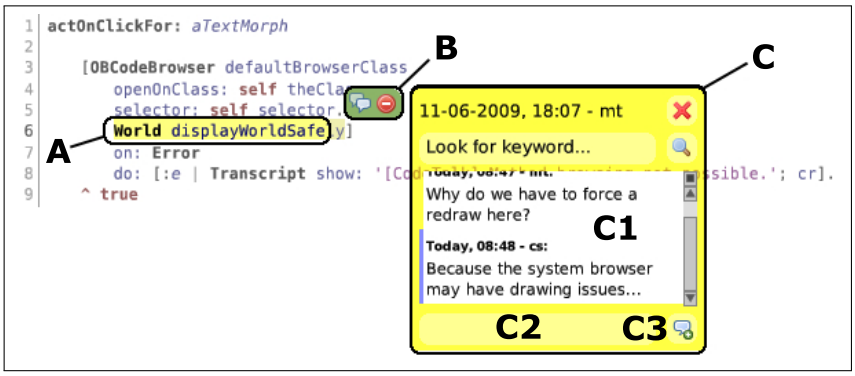

Fig. 1: Code with markup (A), inline morphs (B) and the chat (C)

a markup too. So Dave could talk behind that comment which addresses him and ask further points. The team in Berlin, situation II, would have a defined place to explain their concerns about their solution. The main advantage to an e-mail is the strong connection between the question and the source code. Neither text has to be copied nor its place has to be described. All participants see the same information—e-mails may be deleted already in some inboxes.

Normal syntax highlighting does not judge the source code in any way. However, starting to mark a code snippet red, yellow or green has another intention. Generally, the resulting background color is independent from the programming elements. Developers can evaluate this in many different ways and problems can occur, regarding the team dynamics. User studies will have to show how serious this is and what impact it has to the agile software development process. Anyway, it is possible to change the type of a markup and a discussion about bad code could be toggled to a normal one again.

Studies like [5] about communication behavior in source code comments extracted several types of messages. A *point-to-point* [5] communication addresses a particular person. Even though the whole team should be aware of all implementation details, misunderstandings can happen. New source code is not known by all developers immediately. To support this type of communication, CodeTalk highlights the author's initials of the current Squeak image in text messages. Additionally, the next section will explain the *markup browser* which offers enhanced awareness capabilities for this aspect. *Multi-cast* [5] communication is possible because all markups are visible to everyone who can access the project. Anyway, this should be preferred in an agile team. Task-specific annotations or *bookmarks* [5], that

mention for example places for refactorings or implementation concerns, can be created with a chat or comment markup. That depends on the desire to discuss with other developers.

2.3 Single Point of Information

The markup browser is responsible for showing all markups of a specific scope. This can be a package, a class category, a class or a method category. Figure 2 displays the layout of the browser. At first, markups are filtered (A). The developer can choose between all markups, markups addressed to him or markups created by a particular person. Secondly, the result is sorted and grouped (B). The activity of markups is most important and a proper sorting will be applied when opening the browser. Every annotation has its own list entry and after reading its contents, this entry is grayed-out. The bottom half of the browser is reserved for the annotation contents. A header (C) presents general information like creation time, creator, a hyperlink to the method and the respective code snippet. Below that, all chat messages are listed in reversed chronological order (D)—the latest message comes first.

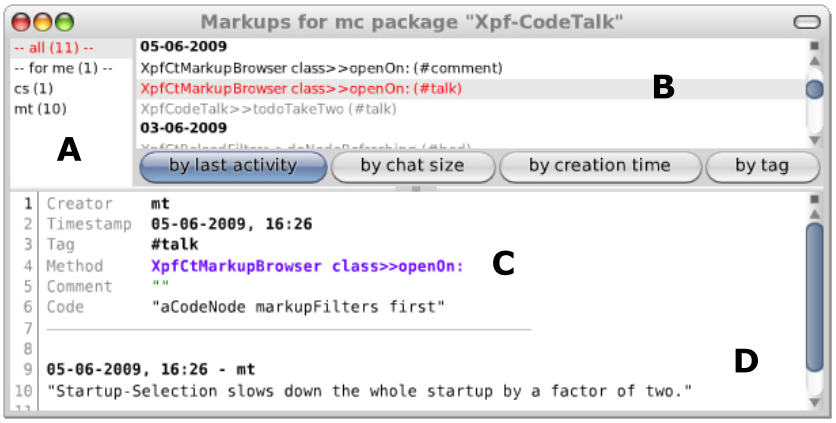

Fig. 2: Markup browser: Provide awareness of new annotations

As the browser opens automatically after a check-out from the repository, Alex and Lucy from situation II could be sure to contact the team in Palo Alto. All concerns about their solution would be noticed when the branch office in Europe is staffed again some hours later. Due to the fact that old annotations are grayed-out, the new ones appear emphasized.

Alex could write down Dave's or Lucy's initials explicitly in the chat to communicate effectively. Those messages determine markups for the filter *"for me"* (A).

2.4 Navigation via Markups

As explained in [6], the *waypoint metaphor* also applies to source code navigation. "A route provides a path from one point to another together with intermediate destinations (a sequence of waypoints)." [6] Such a route through an aggregation of classes and methods can provide a better comprehension when reading new code. Crosscutting concerns and places that share a same background can be connected easily.

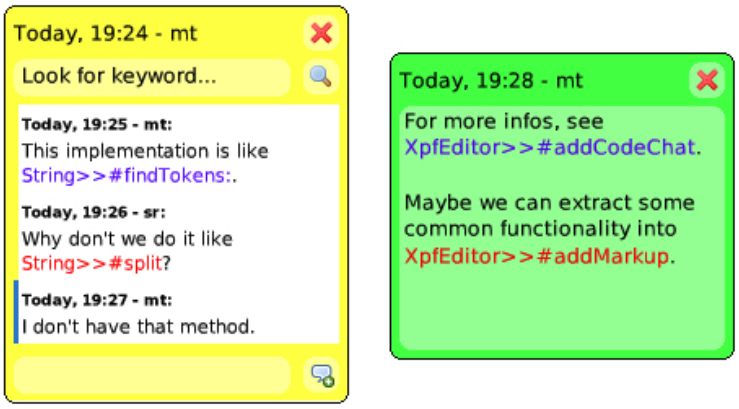

Fig. 3: Method links enable convenient source code navigation

CodeTalk enables the developer to create these waypoints with the help of markups. As visible in figure 3, references to methods will be recognized in chat messages and comments. Hyperlinks will be created and a mouse click opens a new code browser showing that method. If the target is not available in the image, the link appears red. This feature is available wherever the user can access the markup in the image.

To discuss an extraction of a common implementation from several methods into only one method, a markup in one of these methods could point to the other ones. Furthermore, moving content to another place could be described with a markup that points to that place. Whenever one developer talks about a method the other one does not have, both will realize swiftly that they have different code bases. Even routes are

imaginable, that guide a new developer through the main places in the application's source to get a basic understanding from the implementation.

2.5 Preserving the Code

One requirement applies to code access without CodeTalk. An external reviewer may want to take a look at the project's internals but does not have the possibility to use CodeTalk as compatibility problems could make a proper installation impossible. Every project developed with markups must not be dependent on CodeTalk. The code has to be readable and compilable in any case.

Using Monticello 1.5, markups will be ignored silently in a CodeTalk-free image when reading the repository. A check-in would create a version without markups but other developers can browse their local versions with the version browser and restore it again.

3 Implementation of CodeTalk

CodeTalk is developed using Squeak 3.10. Source code versioning is offered by Monticello 1.5. The OmniBrowser framework [3] provides enhanced code browsing facilities and is used to create a markup browser. It comes with the universe [7] package *OmniBrowser-Full version 0.27*. Shout 3.15 enables syntax highlighting and the package *ShoutOmniBrowser* offers compatibility with OmniBrowser.

The following sections explain how CodeTalk is integrated into the given environment. Complex communication flows regarding source code saving and loading are covered in a nutshell to concentrate on CodeTalk and not on Squeak-specific internals. Several class diagrams, that visualize the architecture, use an UML notation with small modifications to support Smalltalk. Therefore, no visibility information will be modeled.

3.1 Source Code as Formatted Text

Commonly, a program's behavior is described in many *lines of code*[3], which can be spread over many files. Descriptive words and phrases between the code can explain points that may be difficult to understand but the application itself does not need these comments to function properly. Modern development environments like Eclipse[4] or Microsoft Visual

[3] Wikipedia: `http://en.wikipedia.org/wiki/Software_metric` (2009-06-29 18:15)
[4] Project website: `http://www.eclipse.org` (2009-06-27)

Studio[5] offer syntax highlighting but the developer knows that the source code itself is stored in plain text without any color or other formatting information.

There are two ways to handle text at runtime in Squeak. The first one is just a collection of characters which is encapsulated as the `String` class and its subclasses. The second one stores additional formatting information besides an instance of `String` and is called `Text`. To avoid confusion between plain text and formatted text, from now on the term *string* addresses instances of the class `String`. Instances of `Text` are simply referred to as *text*.

Each text has *runs* that format the internal string. A run consists of start, stop indices and some attributes like `TextColor`, `TextFontChange` or `TextEmphasis`. These are all subclasses of `TextAttribute`. So it is possible to modify the appearance of a text with `Text>>#addAttribute:from:to:` whereas several runs can share the same instance of an attribute. The sum of all attributes are called *style* of a text.

```
]style[(4 10 7)f2b,f2,b! !
```

Listing 1: Serialized style information

The source code of a method in Squeak is accessible through instances of the class `CompiledMethod`. When reading the *changes file* [8] of the image (e.g. *Squeak3.10.2.changes*), `#getSourceFromFile` guarantees to fetch all style information too. Saving changed source code also includes calling the right methods of a file stream to keep all attributes of the text. These are `#nextChunkPutWithStyle:` and `#nextChunkText` for instances of `WriteStream`.

An example of serialized style information is shown in listing 1. The prefix `]style[` indicates serialized information of text attributes. A list of numbers in parenthesis separated by blanks stores the length of each run in the text. Finally the characters up to `! !` are needed to instantiate the correct attributes. To allocate more than one attribute to a run, a comma stands for a transition to the next run.

The whole preservation of the style is done with help of the `RunArray` class. Each text has one instance that is accessible via `Text>>#runs`. While writing all attribute information onto a stream, `RunArray>>#writeScanOn:` requests each attribute to serialize itself. Therefore `#writeScanOn:` has to be implemented in a subclass of `TextAttribute`. It is important that each attribute has a unique start character (e.g. `f` for `TextFontChange`) and keeps track of the amount of information needed to be recreated. Run arrays are

[5] Product website: `http://microsoft.com/VisualStudio` (2009-06-27)

created by calling `RunArray class>>#scanFrom:` with a stream. The unique start character steers the control flow to `TextAttribute class>>#scanFrom:` of the proper subclass of `TextAttribute`. The run array relies on the attribute to not read more from the stream than necessary. Otherwise the whole image can crash because reading text from the changes file is a very basic and important operation.

```
OBMethodVersion>>source
  | file |
  file := sources at: (sources fileIndexFromSourcePointer: pointer).
  file position: (sources filePositionFromSourcePointer: pointer).
  ↑ file nextChunkText makeSelectorBold "was: nextChunk asText"
```

Listing 2: Reading source code with style for a method version

However, the version browser in the OmniBrowser framework ignores style information from the changes file. To enable text support for all versions of a method, a minor fix was done at `OBMethodVersion>>#source` to send `#nextChunkText` instead of `#nextChunk` to the file stream (listing 2).

Finally there is a specific kind of attributes present—text actions. Instances of `TextAction` offer interaction between the mouse cursor and a text. A use case for this feature may be an embedded link that opens a web browser on a click. This is how *method links* are implemented. Chat messages are supplemented with `XpfCtMethodLink` attributes if a specific pattern is recognized.

3.2 A New Text Attribute

CodeTalk introduces the class `XpfComplexAttribute`—a new attribute for text objects that are in the first place value holders for a code chat. Instances of this class will be called *complex attribute*. Each entry in the chat is stored as a small dictionary with the keys `#author`, `#timestamp` and `#message`. Additionally the creator's *author initials* and the creation time of the attribute are accessible. A tag like `#talk` or `#bad` is used to distinguish different types of a chat visually. Furthermore, there is a field for a classic comment in the attribute. This is just a string which is the point of interest whenever the tag is `#comment`.

In the current architecture, there are no subclasses of a complex attribute present. In the case of adding more elaborate content (e.g. images, audio and video clips) in the future, a common base class could be used to implement essential accessing and serialization methods. Subclasses would realize content-specific needs regarding modification.

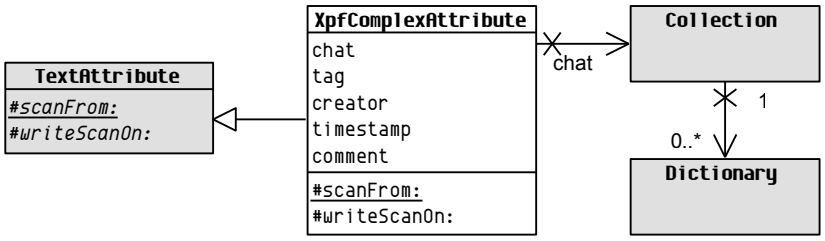

Fig. 4: A new attribute for markups

Obviously, a chat and therefore the content of a complex attribute is of variable length. The characters `h` and `#` surround the string that stores all serialized information (listing 3) to allow `RunArray` to recognize this kind of attribute as well as parsing all suitable data. The `h` was one of the few remaining possible characters and stands originally for *highlight* because an attribute for text background color was implemented initially.

```
h3137204A756E65203230303920353A30373A333120706D-6D74-talk-5B7B226175746
86F72223A20226D74222C202274696D657374616D70223A20223137204A756E65203230
303920353A30373A333920706D222C20226D657373616765223A202248656C6C6F2E227
D5D-6D746D74-#
```

Listing 3: Serialized content of a complex attribute

The content—`timestamp`, `creator`, `tag` and `comment`—will be converted to its hexadecimal representation to avoid collision with control characters in the changes file. The chat is rendered as a *JSON*[6] string and then converted too. Each piece of data is separated by a dash. This conversion doubles the space that each attribute needs.

Whenever new data should be stored in a complex attribute, backwards-compatibility is achieved by appending that data at the end of the list of instance variables and at the end of the stream during serialization. The first one is essential for reliable markup transport with Monticello. As the classic comment was added later, it is located at the end in both places.

An instance of `XpfComplexAttribute` has a fairly short lifetime. Like other text attributes, it is created when reading the source code from the changes file. There should be no reference to that instance when another method's source is displayed. Garbage collection should clean it up then.

[6] Wikipedia: `http://en.wikipedia.org/wiki/JSON` (2009-06-29 18:06)

3.3 Viewing the Code

Each code browser, that uses the OmniBrowser framework [3], has the bottom half reserved for displaying source code. This area is called *definition panel*. In the following, all important classes which are responsible for showing a method's source with style will be described to present the integration points of CodeTalk. All information are derived from class comments and code reading.

Existent Architecture. To show only a small part of the text, a simple `TextMorph` instance is embedded into a scrollable pane called *edit view*. This *text morph* contains a *paragraph* and an *editor* for it. Such a paragraph represents text that has been laid out in some container. It performs drawing routines to display itself in the text morph and the view. The editor is a controller which modifies the paragraph and the scrollpane. The role and detailed usage of the editor will be described in the next section because it is only of minor interest when showing code with markups.

Shout extends this cooperation of view, morph, paragraph and editor with a *styler*. Instances of `SHTextStylerST80` add attributes to the given text object in `#privateStyleText:`. To avoid saving that style into the changes file, `#unstyledTextFrom:` removes all attributes except text actions. The custom view `OBPluggableTextMorphWithShout` contains such a styler to prepare the source code text object. That view class adds Shout functionality to the OmniBrowser framework. The method `#setText:` initiates the styling and `#acceptTextInModel` discards attributes as mentioned before when saving a changed method.

Integration. To integrate CodeTalk, two methods were overridden to use custom subclasses for morph, paragraph, editor and styler (figure 5). The edit view coming with Shout now uses an `XpfTextMorph` instance due to a modification of the method `#textMorphClass`. Such a text morph just defines `XpfEditor` as editor and the `XpfParagraph` class to be used for paragraphs. The pattern of `#*Class` methods, which return a class object to be used, is very helpful to extend the functionality here. The editor adds a reference from a paragraph back to the text morph every time the paragraph is changed (`#changeParagraph:`). While displaying text, the paragraph follows this reference to add, remove or update submorphs of the text morph. The second method that was overridden is `#on:text:getTextSel:accept:readSelection:menu:` to set an instance of `XpfTextStyler` right after the creation of the edit view to handle text styling.

```
XpfTextStyler>>privateStyle: aText
  | backupRuns |
  backupRuns := aText runs.
  super privateStyle: aText.
  backupRuns withStartStopAndValueDo: [:start :stop :attrs |
    (attrs
      detect: [:attr | attr isKindOf: XpfComplexAttribute]
      ifNone: []) ifNotNilDo: [:attr |
        aText addAttribute: attr from: start to: stop]].
```

Listing 4: Save and restore markups when Shout processes code

Before the paragraph is updated with the current text, the styler modifies the appearance of it. Shout removes all attributes of the text it is going to process to ensure a consistent styling. That's why complex attributes need to be copied and restored around `super privateStyle: aText` (listing 4). The second task of the new styler is to keep all markups if the unstyled text is requested. `XpfTextStyler>>#unstyledTextFrom:` does the same thing for complex attributes as `SHTextStylerST80>>#unstyledTextFrom:` does it for text actions which are preserved too. Both kinds of attributes, text actions and complex attributes, are designed to be saved in the changes file because they have semantic importance for the source code.

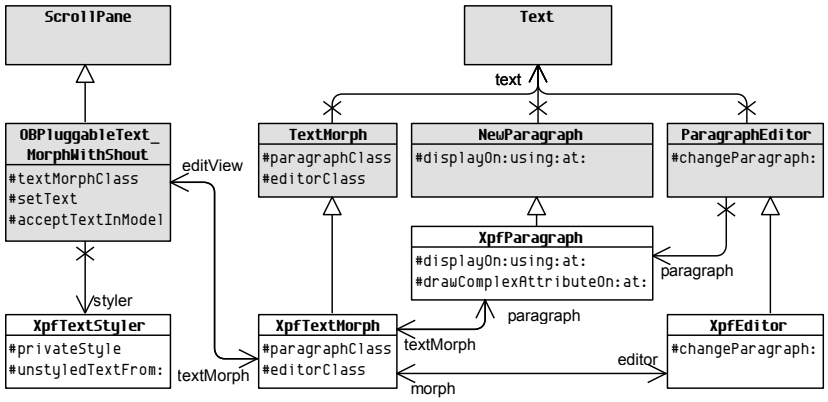

Fig. 5: CodeTalk classes (white) and the environment (gray)

An `XpfParagraph` has updated drawing routines to display the markup. The method `#drawComplexAttributeOn:at:` looks for complex attributes in the given line index and calculates a rectangle for each result. In a final step, it tells the attribute to draw itself on a canvas. This *double-dispatch*

mechanism removes the responsibility from the paragraph to decide how a markup is shown. Merely the calculation of the area is necessary because a complex attribute has no idea which source code it affects. This visualization behind the text is done before the selection rectangle and the text itself are drawn. Markups are visible even for selected text because selections are semi-transparent.

Whenever the text cursor or a whole selection intersects with markups, small icons appear in the upper right of each markup (figure 1). Although this is relevant for editing issues, its appearance will be covered in this section. A complex attribute contains a morph that is a container for several submorphs. This container needs to be coupled with the text morph so that it will not just float around when moving the code browser. Due to the fact that each paragraph has a reference to its text morph, a call of `#addMorph:` is possible. A connection between the text morph and the morph from the attribute can be created. This happens, besides position updates, every time the paragraph renders itself onto a canvas. Obsolete morphs are removed from the text morph with `#removeAllMorphsBut:`. Those containers can quickly become obsolete when the user selects another method or removes a markup.

Creating, adding and moving morphs are very expensive operations. If there would be no change visible to the user, nothing should be done when drawing the paragraph. For this reason, a container morph is only be created once and then stored in the attribute object during its lifetime. Secondly, the text morph ignores an add-request if the submorph is already present. Finally, the movement code is only executed for the last run which has the instance of that attribute. That makes sense because only linked text runs can share such an instance. A visible gap would end in having two instances of this attribute after saving the method. Two container morphs would appear.

3.4 Editing Code with Markups

As described in the last section, the editor is responsible for modifying the text of a paragraph. After each change, `XpfEditor>>#userHasEdited` should be called to tell the text morph that its contents have been updated. Then the user is able to save the changes with the keyboard shortcut *CMD+S* or discard them when browsing another method's source.

Creation of Markups. There are three ways to add a markup to selected source code: press a keyboard shortcut, use the context menu or hit a

button in the toolbar. Every possibility ends up in calling a method of the editor that performs the change. In the case of adding a markup, this would be `XpfEditor>>#tagSelectionWith:` (listing 5). Shortcuts go directly to the editor which receives all keyboard inputs when the scrollable text morph has the focus. They are definied in the class-side method of the editor called `#initializeCmdKeyShortcuts`. Context menu entries are built from subclasses of `OBCommand`. The following methods can be reimplemented to configure the command:

`#label` is mandatory. It returns a string that appears in the menu.
`#execute` is called after the entry was selected.
`#isActive` decides whether the command is visible or not.
`#isEnabled` is used to gray-out the entry and it would be not selectable.
`#group` categorizes entries. Two groups are separated by a horizontal line.
`#cluster` creates a submenu. Entries are created from all subclasses of the implementing class.

There is no statically defined menu. Theoretically, any command is able to appear in all context menus of a code browser if `#cmd*` methods for these commands are implemented in `OBCodeBrowser`. To put commands into the context menu of the text selection, `#isActive` has to examine the current situation. Two instance variables, named `target` and `requestor`, allow access to the circumstances for the context menu request.

In the case of CodeTalk commands, `target` has to be an `OBTextSelection` because markups apply to text. Furthermore, the `requestor` is an instance of `OBEnhancementDefinitionPanel` that has, in contrast to a simple `OBDefinitionPanel`, a reference to the edit view. After the addition of an accessor to navigate from the definition panel to the edit view, the editor got accessible to perform a change.

```
XpfEditor>>tagSelectionWith: aSymbol
  | attribute |
  self startIndex = self stopIndex ifTrue: [↑ false].
  attribute := XpfComplexAttribute forTag: aSymbol.
  attribute addDependent: self morph.
  self text
    addAttribute: attribute
    from: self startIndex
    to: self stopIndex - 1.
  ↑ true
```

Listing 5: Editor adds a markup to the current selection

The third way to add a markup uses the CodeTalk toolbar. `XpfCtToolbar` is a subclass of `OBPanel`. It contains all buttons and is able to call methods of the editor because a reference to the underlying code browser exists. As mentioned before, external navigation is possible traversing the definition panel, the edit view, the text morph and finally the editor. It was decided to prefer the *package browser* whenever code browser-specific method overrides have to be done. So `OBPackageBrowser class>>#panels` was overridden to add support for the toolbar.

Modification of Markups. Complex attributes can communicate with the text morph in the edit view. This is needed to signal changes to the text morph when a chat line was added or the deletion if the attribute is requested via the small inline morphs. The editor enables this communication by adding the morph to the attribute's list of dependents whenever the current paragraph changes (`#changeParagraph:`). There is no need to remove a dependent because the lifetime of an attribute object is very short. By way of example, this mechanism allows an instance of `XpfCtDeleteIconMorph` to request the attribute's deletion on `#mouseDown:` that will be recognized by the text morph (listing 6). More elaborate operations are delegated to and processed by the editor.

```
XpfTextMorph>>update: aSymbol with: anAttribute
  aSymbol = #deleteRequested ifTrue: [
    self text removeAttribute: anAttribute.
    self hasUnacceptedEdits: true].
  super update: aSymbol with: anAttribute.
```

Listing 6: Processing a delete request from a complex attribute

There is a morph to modify the chat: `XpfCtCodeChatMorph`. It appears (figure 1) when the user clicks on the proper inline morph near the markup. This user interface to modify the chat is quite simple. It consists of a morph list for chat messages, some buttons, and two text input fields. When the user adds a new chat message, it is written directly to the attribute. The morph list is updated to show and select the new message. Up to here, no change would be recognized from the text morph. But a new chat message is a change because it is strongly connected with the source code. So the attribute signals this with: `self changed: #chatLineAdded`. The text morph notices this and sets its flag to react later.

3.5 The Markup Browser

The heart of markup-awareness, the markup browser (figure 2), is completely written with the OmniBrowser framework [3]. `XpfCtMarkupBrowser` is a subclass of `OBCodeBrowser`. This brings along automatic updates whenever source code or markups change. Like all browsers, this one has a meta graph to describe the data model statically and a domain graph which is built and changes during the browser's lifetime. Figure 6 shows example graphs for the markup browser using a notation inspired by [3]. Notes with a gray color replace the legend and mention classes, that are used for nodes.

As visible in the left part, markups were browsed for a specific class. This class contains two annotations and none of them is addressed to the author of the Squeak image.

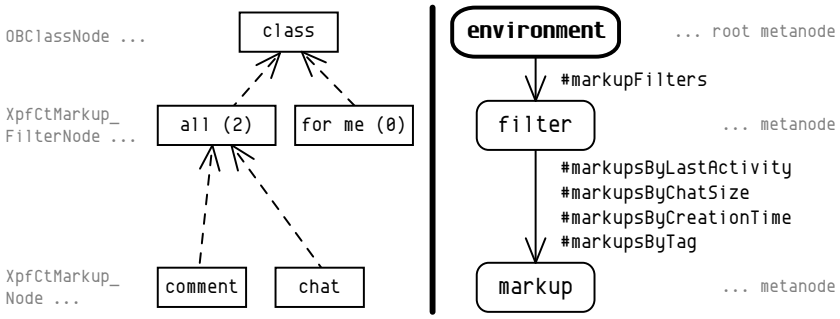

Fig. 6: Markups in a class: domain graph (left) and meta graph (right)

Collecting Markups. The meta graph (figure 6) has three nodes: environment, filter and markup. The first node indicates the entry point. Corresponding domain nodes, that can be used here, are subclasses of `OBCodeNode` and have to implement `#markupFilters` because that is the transition from *environment* to *filter*. CodeTalk supports markup browsing for packages (`OBMonticelloPackageNode`), class categories (`OBClassCategoryNode`), classes (`OBClassNode`) and method categories (`OBMethodCategoryNode`). Usually, the class-side method `#openOn:` is called with one of those code nodes to create a markup browser.

In the strict sense, `OBCodeNode` implements `#markupFilters`. But that implementation calls `#markups` which depends on a code node to supply `#referencesForMarkups` to access the source code. The *filter* node only

accepts `XpfCtMarkupFilterNode` objects. There are four transitions possible to navigate from *filter* to *markup* with differences in sorting and grouping of the resulting nodes. These `#markupsBy*` methods consider activity, chat size, creation time or the attribute's tag.

Showing Content. A domain node (`OBNode`) has to implement `#text` to display read-only contents in the definition panel or `#definition` for contents that are editable. `XpfCtMarkupNode` only needs to implement the first one and returns an appealing version of a text object that contains all data from the markup (see section 2.3 and figure 2).

Markup nodes are grayed-out when the markup was read once in the current Squeak image. `XpfComplexAttribute>>#wasRead` uses a class-side cache, that is a set of arrays, to decide this. Each array represents a unique description of the state of a complex attribute. Reading occurs when the user opens the chat or comment in the system browser and when the markup is selected in the markup browser. In this case, such an array is created and added to the cache but only strings are used for the description.

Performance Issues. To select the filter for *all* markups automatically when opening the browser, `OBCodeNode`markupFilters is called twice more than without startup selection. The problem is, that this call is the most expensive one in this browser. All methods of the root, e.g. a whole package, has to be analyzed to find markups. For transitions between *filter* and *markup* in the meta graph, cached values are used.

At first, markup filter nodes are fetched while navigating from the root node *environment* to *filter*. This is necessary to display at least all filter nodes in the browser. Secondly, the node for the selection is needed. So all markup filter nodes are collected again—this time manually. Finally, the third time, the initial transition needs to be done again when selecting a node programmatically via `OBBrowser class>>#metaNode:root:selection:`. This method creates a browser using the given root meta node, root domain node and selected domain node. That is the way the framework was designed to do startup selection. The *version browser* does it in the same manner.

3.6 Integration into Monticello

A major version control mechanism in Squeak is called *Monticello* [2]. The snapshot of a package, consisting of everal class and method definitions,

combined with some meta data is zipped into an MCZ file and stored in a local or remote repository. To optimize this process, not only all sources are exported into one plain text file named *source.st*, but `MCClassDefinition` and `MCMethodDefinition` instances are serialized to binary data and written into a *snapshot.bin*. In contrast to the changes file of the image, the `MCStWriter` does not write any style information into the *source.st*. Actually, no text is available via `MCMethodDefinition>>#source` at all—only strings.

Method Definitions with Style. There are many methods in the implementation of Monticello that rely on getting a string when requesting code from a method definition via `#source`. Diff, load and merge operations cannot work dependably and may ignore updated source code silently when getting text. So the original idea to store a method's source text in the method definition on creation time was not possible.

Another idea, that is used in the current version of CodeTalk, adds a new instance variable to `MCMethodDefinition`. This variable is called `sourceText`. Method definitions are created from method references which are just lightweight proxies for `CompiledMethod` objects. For this reason, the source code is accessible in its styled form at the creation time of a method definition. `MCMethodDefinition class>>#forMethodReference:` was overridden to fill the new instance variable. Now the source text is exported into the *snapshot.bin* along with the whole method definition object. In the end, versions of the markup are stored in the repository.

Only the binary representation of the snapshot is needed to get the markup into an image after loading or merging a new version from the repository. Commonly, the *source.st* will be ignored. Method definitions are integrated with the help of a `MethodAddition`. Another override of `#addMethodAdditionTo:` in `MCMethodDefinition` allows to transport the markup from the repository into the image. Here, `sourceText` is read from the definition instead of `source`.

```
MCMczReader>>loadDefinitions
  definitions := OrderedCollection new.
  (self zip memberNamed: 'snapshot.bin') ifNotNilDo: [:m |
    [↑ definitions := (DataStream on: m contentStream) next definitions]
      on: Error do: [:fallThrough ]].
  "otherwise"
  (self zip membersMatching: 'snapshot/*')
    do: [:m | self extractDefinitionsFrom: m].
```

Listing 7: Integrated error handling for a "broken" snapshot.bin

Loading without CodeTalk. A developer, who does not have CodeTalk installed, can load or merge a new version safely. Monticello 1.5 has already a fall-through mechanism integrated (listing 7) to handle a "broken" *snapshot.bin*. An image with an unprepared `MCMethodDefinition` class cannot hold the source text because one instance variable is missing and therefore deserialization fails. That is no problem because the *source.st* contains the same source code as the method definitions. Only the markups are missing.

3.7 Overrides and Compatibility

Several methods had to be overridden to create necessary entry points for CodeTalk. During development, an internal goal was to need less overrides as possible. There were some patterns in the given environment which reduced this need to a minimum, for example using `#*Class` methods (see section 3.3).

The most important overrides will be described in the following to illustrate what can get lost after upgrading the OmniBrowser framework or Monticello.

`RunArray class»#scanFrom:` The class is categorized in *Collections-Arrayed* and updates occur rarely. The override is needed to recognize a complex attribute on a given stream and create an instance of that attribute. When reading the changes file, this method is called.

`OBPluggableTextMorphWithShout»#textMorphClass` The package *ShoutOmniBrowser* extends some classes of the OmniBrowser framework with Shout functionality. If the framework does not use another pattern to choose a text morph, it is more likely that an updated Shout extension will override it again. CodeTalk needs this to integrate its own text morph, editor and paragraph classes.

`OBPluggableTextMorphWithShout class»#on:...` Like `#textMorphClass`, an updated Shout extension may use its own `SHTextStylerST80` again. Then there would be no markup visible to the user. All complex attributes would be removed from the text before writing it into the changes file.

`MCMethodDefinition class»#forMethodReference:` An update of Monticello may use other creational methods or override this one again. The source text with markups would be ignored. A new class definition could drop the instance variable for source text. This method is responsible for storing the method's source with markup into the *snapshot.bin*.

`MCMethodDefinition»#addMethodAdditionTo:` If this method is reverted to its origin, the source text from the method definition would be ignored.

No markup transport from the repository into the changes file of the image would happen.

`OBMethodVersion»#source` This is needed to read a method version with style information from the changes file. Otherwise the *version browser* does not show any markups.

A new version of `XpfComplexAttribute` may introduce new data. Old ones can still exist in the changes file or a Monticello repository. Being backward compatible with the changes file depends on the serialization algorithm. That's the reason why new fields are added to the end of the stream (listing 8).

Instance variables of a class are numbered serially. When importing an object from the *snapshot.bin* its state is transfered by copying one value after another. The name of a variable is unimportant. Only its position matters. Therefore, new data fields have to be added to the end too.

```
XpfComplexAttribute>>writeScanOn: strm
  strm nextPut: $h.
  self
    writeTimestampOn: strm;
    writeCreatorOn: strm;
    writeTagOn: strm;
    writeChatOn: strm;
    writeCommentOn: strm. "new data field at last"
  strm nextPut: $#.
```

Listing 8: Serializing a complex attribute on a stream

Being forward compatible is of minor interest but mentionable regarding Monticello. It can happen that markups of a newer version are loaded into an image that has an out-dated `XpfComplexAttribute` class. Usually, the fall-through mechanism of Monticello will discard the whole method definition in this case. The current implementation of CodeTalk does not change this behavior. A solution would be to reimplement `Object>>#instVarAt:put:` for complex attributes (listing 9). There was no time left when writing this thesis to try it out because extensive testing is important for such a basic operation.

```
XpfComplexAttribute>>instVarAt: index put: anObject
  ↑ index > self class instVarNames size
    ifTrue: [anObject "or nil?"]
    ifFalse: [super instVarAt: index put: anObject]
```

Listing 9: Idea for adding forward compatibility to markups

New variables would be ignored but the method definition itself would be processed. It wouldn't be perfect but would have had the advantage to keep some markup information. Nevertheless, all developers should have installed a version of CodeTalk that is up-to-date.

4 Case Study and Evaluation

All design decisions and functional concepts are based primarily on books like [1], papers [3] [5] [6] [9], suggestions from involved developers and the own experience so far. Nevertheless, real user testing is important for every software project. In this case, all agile development teams form part of the target group. The current implementation of CodeTalk constraints that a project has to be developed in a Squeak environment.

Actually, there were 80 students who got the chance to make use of CodeTalk in their projects. In the lecture *Software Engineering I*, these students formed 16 different teams to develop applications with database access in Squeak. An agile software development process like *Extreme Programming* [1] had to be used. Basic knowledge about the programming language Smalltalk could be assumed and a prepared image with CodeTalk was provided. Summing up, all prerequisites for were met. The project's time frame was about three months. After two-thirds of that time, sources of all groups were analyzed for markups and personal interviews offered valuable insights.

Fortunately, markups were used—in the first place to write down tasks that need to be done. This included planned refactorings of bad source code and new features that needed to be implemented. The need of asynchronous communication was present because it was often late or the developer was too lazy to take a look at its instant messaging client. Using CodeTalk seemed to be fastest way in that situation. All addressed persons, the whole team in most cases, noticed those annotations. The markup browser was judged to be clearly arranged and helpful. Surprisingly, there were no problems with colors regarding the team dynamics. The meaning was obvious to most students: Green was supposed to be a comment, red marks bad code and yellow was like a memo for the whole team. Finally, *self-communication* [5] occured while comment markups were used for personal notes because there is only one free text field.

Other opinions offered problems, for instance when adding a markup to describe a new feature. No source code may be available to be used for. Unfortunately, the auto-completion capabilities of *eCompletion*, an extension package for Squeak, removed markups sometimes. Regarding

colors, a red annotation could have the same topic inside like a yellow one. Then color had no effect to a student's behavior at all. Surprisingly, method links were not used because nobody knew they even exist. Somebody found the appearance of the markup browser after each merge operation in Monticello very annoying. The icon for resizing a markup, a paint bucket, was not recognized for performing this change. At that time, no long code chat evolved because the creator only waited for the markup to disappear and no further participation in the chat happened.

Some teams did not use CodeTalk because they were working together all the time. In that case, there was no need for effective asynchronous communication. The general impression was that e-mails are becoming unnecessary when talking about source code is possible with CodeTalk.

5 Related Work

Almost 20 years ago, tools like *ICICLE* [9] were developed to add annotations to source code. *Code inspection* meetings were used to read and talk about code. The purpose was to optimize the system, discover bugs and discuss other concerns. Resulting *stand-off* markups needed a tool to visualize them together with the source code. Every line could get an annotation with ICICLE. Small icons at the beginning of each line indicated their presence.

A more recent example tries to combine *waypoints* and *social tagging* in *Javadoc*[7] comments: *TagSEA* [6] [10]. The current version 0.6.6 was released[8] in October 2008. There is no new idea in presenting the tags in the code. They are visible as normal Javadoc comments but all extra information is 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 [10] 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. Actually, not only Java is supported by TagSEA because bigger projects rarely use only one programming language. This support means the detection of annotated programming elements like methods, classes or packages.

CodeTalk uses stand-off annotations like ICICLE did to avoid distraction. Looking at possible scopes, it offers a more fine-grained annotation

[7] Project website: `http://java.sun.com/j2se/javadoc` (2009-06-27)
[8] Project website: `http://tagsea.sourceforge.net` (2009-06-27)

than just whole lines but TagSEA recognizes a good deal more programming elements. However, an intensive use of Javadoc comments could make it hard to read the source code itself. 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.

6 Conclusions and Future Work

Regarding the vision of the introduction section and all results of the case study, several open tasks follow:

Scope. At the time, CodeTalk supports only markups for code snippets of a method. Chats for whole methods, classes, class comments or packages are imaginable.

Content. The vision claims to share not only text messages but audio files and video clips (section 1.2). Sophisticated serialization and versioning mechanisms have to be used because that kind of data would have a bigger extent.

Persistence. Serialization of the code chat and other information of the complex attributes could be optimized to use less space in the changes file. Another way than using the *snapshot.bin* in a Monticello Zip archive is conceivable. An own *style.bin* could store all data to rise compatibility with CodeTalk-free Squeak images.

Extensibility. A generic interface to other systems like *ProjectTalk* [**?**] would make it possible to store any kind of data in an annotation. This could be a reference to an open task or *user story* [1].

Awareness. Nicknames or the full *author name* of the image are more likely to be used in a chat message which is addressed to a person. The creator of a markup could get a notification after loading the source code if its annotation was read by the target person.

Although more ideas exist to optimize and extend CodeTalk, agile teams can already benefit from this convenient, effective mechanism in its current release. When doing asynchronous communication about source code in this way, there is almost no need for e-mails to perform that task.

References

1. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1999) 1, 2, 3, 4, 23
2. Colin Putney: Monticello - A distributed optimistic concurrent versioning system for Squeak code. Website: `http://www.wiresong.ca/Monticello` [Online; accessed 2009-06-27]. 4, 19
3. Bergel, A., Ducasse, S., Putney, C., Wuyts, R.: Meta-driven browsers. Lecture Notes in Computer Science **4406** (2007) 134 4, 9, 13, 18, 23
4. Squeak Community: Morphic. Squeak Swiki: `http://wiki.squeak.org/squeak/30` [Online; accessed 2009-06-27]. 5
5. Ying, A.T.T., Wright, J.L., Abrams, S.: Source code that talks: an exploration of eclipse task comments and their implication to repository mining. SIGSOFT Softw. Eng. Notes **30**(4) (2005) 1–5 6, 7, 23
6. Storey, M.A., Cheng, L.T., Bull, I., Rigby, P.: 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) 195–198 8, 23, 24
7. Squeak Community: Package Universes. Squeak Swiki: `http://wiki.squeak.org/squeak/3785` [Online; accessed 2009-06-27]. 9
8. Squeak Community: .changes file. Squeak Swiki: `http://wiki.squeak.org/squeak/49` [Online; accessed 2009-06-27]. 10
9. Brothers, L., Sembugamoorthy, V., Muller, M.: 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) 169–181 23, 24
10. Storey, M., Cheng, L., Singer, J., Muller, M., Myers, D., Ryall, J.: How Programmers can Turn Comments into Waypoints for Code Navigation. In: IEEE International Conference on Software Maintenance, 2007. ICSM 2007. (2007) 265–274 24