Creating Guided Code Explanations with chat.codes

STEVE ONEY*, The University of Michigan, USA
CHRISTOPHER BROOKS, The University of Michigan, USA
PAUL RESNICK, The University of Michigan, USA

Effective communication is crucial for programmers of all skill levels. However, communicating about code can be difficult, particularly in asynchronous settings where one user writes an explanation for another user to read and understand later on. Communicating about code is uniquely difficult for two reasons. First, code-related explanations are dichotomous, containing fragments of code and associated natural language descriptions that are not necessarily sequential. Second, instructions' explanations of code often involve multiple stages of modifying code in steps throughout their explanation, but these intermediate steps are difficult to capture. This paper introduces chat.codes, a new tool for creating guided explanations about code, meant to be consumed asynchronously. chat.codes introduces three features that make it easier to communicate about code. First, it introduces deictic code references, which allow users to reference specific region of code in parts of their explanations. Second, it tracks and summarizes code edits in-line with messages, allowing users to create explanations in stages. Third, it tracks every version of code, enabling future users to back-track to previous version of code to reconstruct the context for code references. An evaluation showed that these features were beneficial for both instructors and students in an introductory programming course.

ACM Reference Format:
Steve Oney, Christopher Brooks, and Paul Resnick. 2010. Creating Guided Code Explanations with chat.codes. *ACM Trans. Web* 9, 4, Article 39 (January 2010), 20 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Communication is a fundamentally collaborative act where two (or more) parties try to reach "common ground" [14, 15]. Communication about code poses several unique challenges. First, communication is typically about specific parts of the code, and it is difficult to establish common ground about which part of code is being discussed. Second, the code being discussed may change during the conversation, which complicates any reuse of the conversation history, either within the session or at a later time.

We developed a tool called chat.codes to address some of the challenges of discussing code. As with some other tools for communicating about code [13, 28], it provides side-by-side windows for code and natural language explanations. Its novel contribution is the integration of three other features: deictic code pointers, code versioning, and inline code diffs. Pointers allow for chat messages to reference to specific segments of code. Versioning maintains a version history for the code, and ties the chat contents (including code pointers) to specific versions. Diffs are automated summaries, in the chat window, of code changes between versions of the code.

*This is the corresponding author

Authors' addresses: Steve Oney, The University of Michigan, 105 S State St. Ann Arbor, MI, 48103, USA, soney@umich.edu; Christopher Brooks, The University of Michigan, 105 S State St. Ann Arbor, MI, 48103, USA, brooksch@umich.edu; Paul Resnick, The University of Michigan, 105 S State St. Ann Arbor, MI, 48103, USA, presnick@umich.edu.

1559-1131/2010/1-ART39 $15.00
https://doi.org/0000001.0000001

chat.codes can be used in several alternative workflows. One is synchronous chat (where participants are involved in the conversation at the same time), sharing the code and chat windows between multiple remote users. It can also be used asynchronously, where one user writes an explanation meant to be read later (either by a single user or multiple users). The two modes can also be combined, with people dropping in and out of participation and with later participants "catching up" on the conversation and possibly adding to it.

In this paper, we describe the tool and report on the first stage of evaluating the tool, assessing whether it is usable and useful in the simplest (but arguably one of the most important) workflows — in creating *guided explanations*, which we define to be a one-stage asynchronous communication where an instructor constructs an explanation for a chunk of code and learners consume that explanation later on. In particular, in a qualitative study with four experienced programmers, they quickly learned to use the features to create guided explanations. In a second study, nine students from an introductory programming course were each exposed to both conventional text explanations and explanations that take advantage of the pointers, versioning, and diff features of chat.codes; all of them preferred the chat.codes explanations. In a third study, twenty-four freelancers with some programming experience were randomly assigned to use either chat.codes explanations or video-based code explanations with the exact same content; performance with chat.codes was better on average, but the results were not statistically significant.

In summary, the main contribution of this paper is a description of the key features of chat.codes and some of the lessons learned from the design iterations. The user studies serve to validate that the features are indeed usable and useful. In particular, design contributions include:

(1) a markup language for making explicit references to code, and an interaction paradigm for creating and editing them via mouse operations in a code window;
(2) a way to link the code version history and chat log, so that the chat log can be used to navigate the code history; and
(3) a feature that automatically inserts summaries of diffs between code versions into the chat log.

2 RELATED WORK

chat.codes builds on prior work in HCI, CSCW, and Software Engineering.

2.1 Improving Collaboration with Deictic References

Effective communication in collaborative tasks requires grounding conversation through shared context [14, 15]. *Deictic* expressions reference the time, place, or situation in which someone is speaking. For example, someone might say "over *there*" or "*that* one"; the meaning of these references depend on a shared context. chat.codes uses deictic references to ground chat messages with code context. Deictic references have been studied and used to improve communication in a variety of other contexts, including remote collaboration in shared workspaces [23] and on physical tasks [16], where it has been shown that they can help communication by grounding messages in a shared context.

Korero [10] allows users to make deictic references to point to course-related materials (such as a specific time in a video). Although Korero also works with documents, it assumes that the content of the materials being referenced is static, an assumption that does not hold with many code discussions, where the code content dynamically changes. chat.codes instead works with code that evolves throughout the course of the discussion while ensuring that code references never become stale.

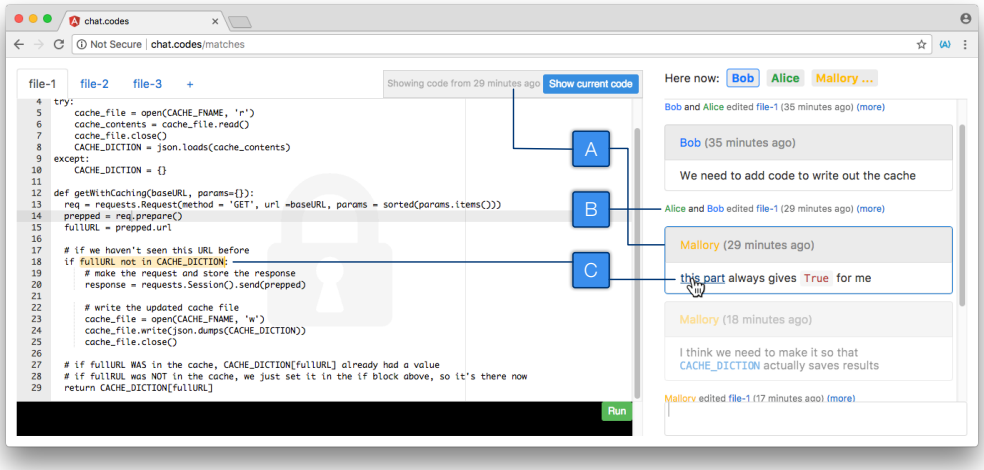

Fig. 1. chat.codes is a communication tool for programmers that combines a shared editor (left) with synchronous text chat (right). Users can click on any given message to see the state of the code when that message was sent (A). chat.codes also allows programmers to see when code was edited by displaying edit messages in-line with chat messages (B). These messages can be expanded to reveal a full diff (Figure 5 shows what clicking the '(more)' button displays). Users can also easily "point" to regions of code through a simple highlighting interaction (as Figure 3 illustrates).

Annotations — notes or comments that are attached to a shared workspace — are one of the most common form of deictic references in computer interfaces. Many commercial collaborative document editors, including Google Docs and Microsoft Word, allow users to annotate content via comments that are attached to specific portions of text. Annotations have also been explored in the context of multimedia artifacts [44], 3D space [40], and online course materials [47].

Annotations have also been explored for code explanations. WebEx [5] allows programmers to annotate lines of programming examples. Although annotations can be helpful in associating a comment with its relevant location, they are not as useful for guided code explanations or synchronous or asynchronous discussion. This is because whereas the natural order of annotation consumption matches the order of the text in a document (from the first line to the last), guided code explanations often involve a walkthrough that is non-sequential with respect to the code being discussed. Further, annotations can be difficult to interpret in the context of evolving code, where it is unclear if an annotation refers to an old version of code or the current version. chat.codes addresses both of these problems when creating guided code explanations by enabling code pointers to be part of a larger coherent explanation that does not need to be sequential and can involve multiple code versions.

2.2 Tools for Discussing Code

Other researchers have recognized the need for programmers to be able to communicate with each other and have created tools to allow them to do so. Prior work in this area can be divided into synchronous and asynchronous tools.

2.2.1 Synchronous Code Chat Tools. Many programming Integrated Development Environments (IDEs) and tools have integrated synchronous discussion mechanisms as a way to allow remote programmers to communicate. All of these tools build on shared editors that allow any number of participants to edit code simultaneously. Some of these tools enable text-based chat between participants [20–22, 27, 42]. Others offer video and voice communication [13, 29]. As the

introduction describes, chat.codes goes beyond these tools by tightly coupling messages with the code they are discussing. This allows conversations to be re-contextualized by future readers.

Prior research has also explored additional communication channels for remote communication between programmers. D'Angelo and Begel found sharing gaze information from eye trackers could improve communication in remote collaborative programming sessions by making it easier to communicate about locations in code [11]. Although this can help resolve ambiguity, it requires specialized hardware (an accurate eye tracker). Further, although the ambiguity and error rate that is inherent to using gaze to infer a code location might be acceptable in synchronous discussion settings (where ambiguity can be resolved through discussion) but might be less acceptable for explanations that could be read asynchronously, as chat.codes explanations and discussions can.

2.2.2 Asynchronous Discussion Tools. There are also many asynchronous communication tools for programmers. Despite being less natural than synchronous communication, asynchronous communication tools have the benefit of easing coordination costs between discussion participants, as they do not need to be involved at the same time. Ginosar et al. proposed a technique for creating asynchronous multi-stage code examples [18]. This system helps authors to construct a series of code examples that build to a complete solution, propagating changes in one code version forward or backward to other versions. However, it does not provide a way for text explanations to link to code versions or particular parts of them, as chat.codes does.

Many programming support tools rely on asynchronous discussion forums to provide scalable support [3, 30, 36, 41]. These discussion forums are useful even for programmers who do not participate in a discussion because they build "accumulated knowledge" [1, 2, 32], meaning programmers can find answers to their questions by looking at previous discussions. Previous studies have shown that *most* of the usage of discussion forums come from reading answers to previous questions [4, 26, 33, 35].

In order to build accumulated knowledge through posts, many forums define strict posting guidelines [32]. Stack Overflow, for example, has a set of guidelines and recommendations on how to ask a "good" question. According to these guidelines, good posts should contain concise and accurate descriptions of the problem and often should include a "minimal, complete, and verifiable example" [36]. For novice and learning programmers, however, it can be difficult to summarize a problem or to know which portions of their code base might be problematic. Such users benefit from synchronous discussions [7] where they can walk through their problem with the help of a remote helper [25]. chat.codes aims to provide natural synchronous communication while also allowing previous discussions to be useful for other programmers.

2.2.3 Blending Synchronous and Asynchronous Support. Previous researchers have proposed blending the best features of synchronous and asynchronous communication [37]. Codeon [7] uses a "semi-synchronous" communication model where programmers ask questions using synchronous mechanisms (voice and text) but remote helpers respond to requests asynchronously. However, Codeon's asynchronous model relies on using different types of communication for the programmer (who uses voice) and remote helpers (who use annotations, code changes, and textual explanations). In this model, developers cannot tie textual explanations to regions of code as they can in chat.codes and helpers must rely on annotations to point out specific locations in code.

2.3 Version Control Systems

GitHub and other Version Control Systems (VCSs) tools enable formal discussions of code. Professional programmers often use commit messages, pull requests, and issue trackers to communicate with collaborators. However, VCSs are often too heavy-weight for explanatory discussions because they require actively committing and labeling code changes. Azurite [45, 46] and CodePilot [42]

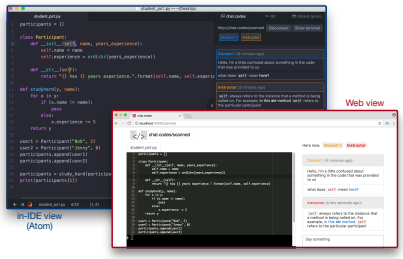

Fig. 2. chat.codes is implemented as a website and as a plugin for the Atom IDE. Atom users can use chat.codes in their IDE, which will automatically converts their editor into a shared editor. Programmers can click a button to create a unique shareable URL that helpers can then visit. Helpers can then access the requester's IDE over the web by visiting that URL.

have explored ways of making the features of VCSs more lightweight. Azurite provides a visual timeline view to explore the history of code changes. CodePilot [42] included the automatic posting of notification of version control events such as new commit messages into a chat log, a feature that some software development teams also include in their use of the commercial product Slack. Prior work, however, has not investigated how to inject useful summaries of code diffs into the chat stream.

2.4 Improving Scalability for Programming Support Tools

One of the benefits of allowing conversations to be re-used is that it helps create more scalable support. Previous work has explored alternative ways to provide more scalable support for programming courses. Codeopticon [21] improves instructors' ability to monitor many students in real-time to detect when students need assistance in the context of large courses. When an instructor detects that a student needs assistance, they can synchronously chat with that student. OverCode [19] enables more scalable programming support by allowing instructors to group student's solutions to programming exercises and allowing instructors to give feedback to multiple students simultaneously.

3 CHAT.CODES DESIGN

Our design goals for chat.codes's interface were:

- To *minimize its learning curve* by ensuring that all of the same interactions and conventions in standard chat tools[13] also work with chat.codes,
- to *prevent the kinds of information overload* that advanced change-tracking systems (like VCSs) can introduce, and
- to *reduce the burden for programmers to provide context* by automatically tracking context when possible (which is crucial to allowing programmers to later reconstruct this context).

chat.codes is implemented as both a standalone web application and as a plugin for the Atom [28] IDE, as Figure 2 illustrates. Users can create chat channels from either interface. Both interfaces have the same interactions, but the figures in this paper and evaluations use the web interface.

chat.codes includes a multi-user code editor and a text-base chat interface. Like other shared editors, users can see other users' cursors and cursor selections in real-time, simultaneously edit the same code, and chat in real-time. Users can create a new discussion channel by visiting https://chat.codes/ and clicking "New Channel". This generates a one-word channel name, which

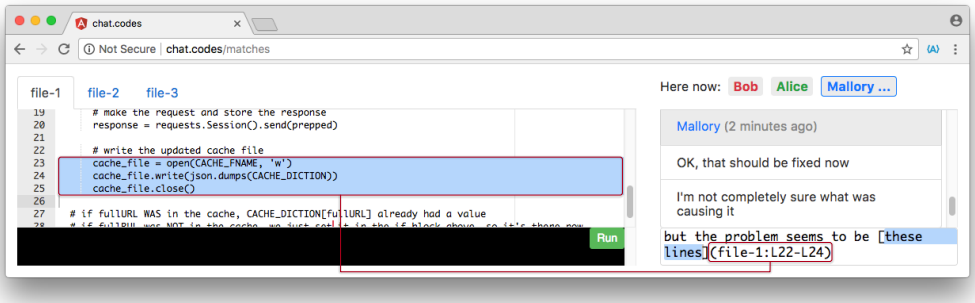

Fig. 3. chat.codes enables programmers to point to regions of code in their messages through a Markdown-based syntax. Users can also easily specify what region of code they want to refer to by simply highlighting that region in the code window, which automatically inserts an appropriate code pointer.

other users can join by visiting https://chat.codes/*[channel name]*. Every channel name is a random short English word, to make it easy to verbally share a channel name.

3.1 Code Pointers

One novel interaction that chat.codes introduces is "code pointers", which allow programmers to add deictic references to regions of code in their messages. For example, in Figure 1C, Mallory writes "this part always gives `True` for me". When another programmer hovers over "this part", the region of code that Mallory is referring to is highlighted (Figure 1C). Code pointers allow programmers to create code references that never become invalid, even if the region of code they point to is later modified or removed.

In other shared-context chat tools (e.g., Google Docs, [13, 27]) when users want to point to a region of a document, they must send their message while highlighting the relevant region and assume that the remote collaborator is paying attention at the right time. However, this convention does not allow outside observers to determine what region of code a given user is referencing from the chat log alone. Further, users cannot ensure that the message recipient actually saw the region of code they highlighted.

Some tools also allow users to add annotations or inline comments associated with regions of code. However, these are not a substitute for code pointers because: 1) they are not temporally linked with the users' discussion, meaning that step-by-step explanations that walk programmers through distributed portions of code are not practical with inline comments, 2) they can be invalidated if the region of text associated with the inline comment is changed or removed, and 3) code pointers allow users to reference multiple regions of code in the context of a single message whereas inline comments require one message per code region.

3.1.1 Creating Code Pointers. We use a syntax for code pointers that is based on Markdown [34] links: `[text](URL)`. chat.codes uses three subcategories of code pointers for:

- an individual line: `[this line](code.js:L24)`,
- a range of lines: `[these lines](code.js:L23-L42)`,
- or any range: `[this code](code.js:L24,0-L24,3)`

However, we felt that requiring users to enter code pointer manually was unnecessarily difficult and unnatural. Thus, chat.codes also allows programmers to write code pointer by simply selecting

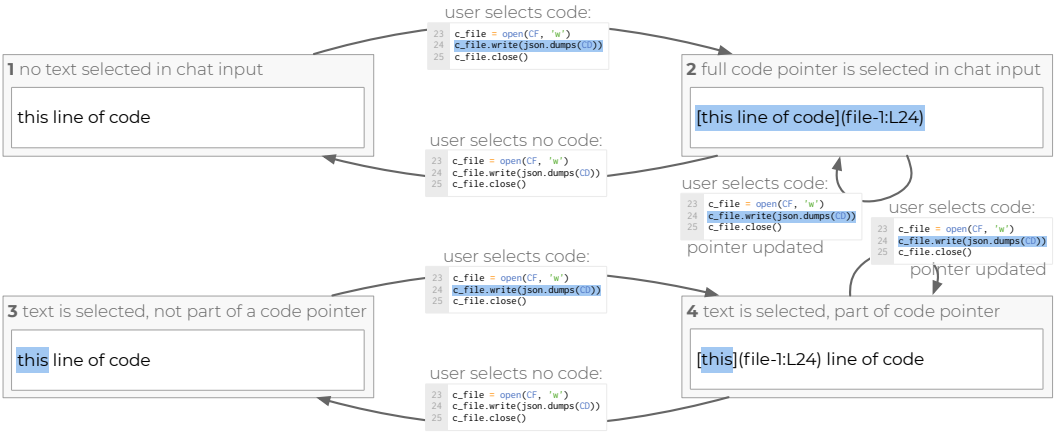

Fig. 4. In order to allow users to easily create code pointers in chat messages, we designed a set of interactions that depend on the state of the chat input box. These interactions are described in more detail in section 3.1.

regions of code in their editor (see Figure 3). Effectively designing this feature was surprisingly challenging because a naïve approach would result in users' typed messages being overwritten whenever they selected code. Through a rounds of pilot testing with users, we arrived at a nuanced set of interactions that depend on the state of the chat input box. These states are illustrated in Figure 4 and described below:

(1) *no text is selected in the chat input box (or the chat input box is empty):*
When the user selects a region of code, the system will insert a new code pointer that points to the user's code selection and whose text is the first 10 characters of their code selection. Then, it will select the full code pointer text (putting them in state 2).

(2) *a code pointer is selected in the chat input box:* `[this line of code](file-1:L24)`:
When the user selects a region of code, the system will replace the selected code pointer with a new code pointer that points to the user's new code selection and whose text is the first 10 characters of their new code selection. Then, it will select the full code pointer text (stay in state 2). If the user has an empty code selection, then remove the complete code pointer (go back to state 1).

(3) *text is selected but it is not part of a code pointer:* `this line of code`
When the user selects a region of code, the system will add a link that points to the selected code and whose text content was the word that was previously selected in the chat input box. Then, it will select the same text in the chat input box. In the above example, "`this`" would remain selected but a code pointer would be added around it (putting them in state 4).

(4) *text is selected that is part of a code pointer*[1]*:* `[this](file-1:24) line of code`
When the user selects a region of code, the system will update the code pointer but not the text. The previously selected text also remains selected (remain in state 4).
If the user has an empty code selection, then it will remove the code pointer portion while keeping the selected chat input text (putting them back in state 3).

Despite the complexity of these rules, we found that they were intuitively understandable with a minimal learning curve for users in our evaluation. To illustrate why, consider the following interactions:

[1] This also applies for selections that partially overlap code pointers (e.g., "`[this line of code](file-1:L24)`")

- Alice wants to insert a new code pointer into an empty message (state 1). She intuitively starts by highlighting code (putting her into state 2). As she continues to drag her code selection, the code pointer in the chat window updates. If she clears her code selection, the code pointer is removed, which leaves the content of the chat input box as it started (back in state 1).
- Bob wants to add a code pointer to a portion of his message. He intuitively highlights the portion of the message he wants by selecting "this" in "what does this mean?" (state 3). As he selects code, a code pointer is automatically inserted "this" (putting him in state 4). If he clears his code selection, the pointer is removed, which leaves Bob where he started: with "what does this mean?" selected (back in state 3).

3.1.2 Code Pointers to previous versions of code. chat.codes's code pointing feature benefits from (and integrates with) its other features. Specifically, the fact that chat.codes allows users to navigate to previous versions of code allows code pointers to remain "valid", even if the portion of code it links to is later removed or if the file it references is deleted. Every code pointer is associated with the specific version of code when the code pointer was created. If a user creates a code pointer and that code is edited later on, the original version of code (when the pointer was created) is restored when that code pointer is referenced. Sections 3.2 and 6 discuss this in further detail.

3.2 Navigating Through Time and Code

Per our goal of minimizing its learning curve, by default, chat.codes behaves like every other shared text editor with integrated chat; users can send messages and see the current state of code through a shared editor. However, unlike previous code discussion tools, chat.codes also allows programmers to navigate to earlier versions of code by clicking on messages or edits (see Figure 1A). This allows them to easily reconstruct the context of a given message.

Users can navigate through the code history in two ways:

- by clicking on any message, which will show them the state of code when that message was sent or
- by selecting a code pointer, which restores the version of code with which the pointer is associated and highlights the region of code that it references, allowing users to see the full context in which that code was referenced.

There is a danger that navigation through code versions will lead to confusion, both to other users sharing the code editor and even to the navigator. We made two design design choices to minimize that potential confusion. First, when a user looks at a previous version of code, that user enters into a non-shared mode; it does not affect the code view of other users. Second, the non-shared editor is set to read-only, indicated by the subtle lock in the background of the code window in Figure 1. This prevents the user from making changes without realizing that the changes will not be visible to others. The code can still be executed, and elements can be copied for pasting back into the shared live version when the user returns to it. To ensure the navigator understands what version of code is shown, chat.codes highlights the message they clicked, greys out messages that were sent later, and displays a message above the code window with information about what code version they are looking at and a button that allows them to return to the current code version. For example, in Figure 1, Bob is looking at a version of code from 29 minutes ago, which he accessed by clicking Mallory's message.

3.3 Summarizing Code Diffs and Edits

chat.codes displays code edits as status notifications in the chat window (see Figure 1B). Code edits are "grouped" together if they occur sufficiently close in time (no longer than 5 minutes between the first to last edit) and if there are not chat messages between edits. By default, these

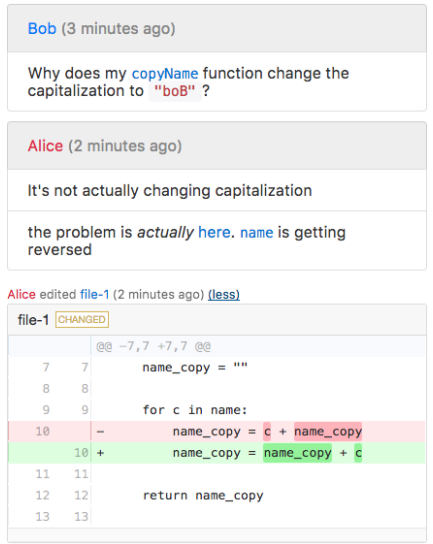

Fig. 5. In chat.codes, code edit summaries are displayed inline with chat messages. By default these summaries only indicate which user changed which files. They can also be expanded to reveal a full diff, as shown above.

diff summaries are concise and only display information about who edited which files when (e.g., "*Alice* and *Bob* edited file-1, file-2, and file-3 (3 minutes ago)"). However, they can also be expanded to reveal more information about what edits were made, as Figure 5 illustrates.

In our evaluation, we found that some programmers would leverage these inline code diffs by interleaving code edits with explanatory messages. By doing this, they create a step-by-step explanation of their code edits. For programmers looking at earlier discussions, these diffs help them understand how code is changing over time.

3.4 Code Execution and Results

The interface for chat.codes allows users to test and execute a their code using a terminal (in the bottom left of Figure 1). Unlike the code editor, this terminal is not shared between users, a design decision we made after recognizing that programmers might want to execute different versions of code (for example, if a programmer wants to run an older version of the code).

4 EVALUATIONS OF CHAT.CODES FOR WRITING AND READING EXPLANATIONS

In evaluating chat.codes, we wanted to understand its ability to handle different aspects of explaining code. To this end, we carried out three studies to determine 1) the ease of explaining concepts using chat.codes, 2) the usefulness of code references and edit history, and 3) to compare explanations in chat.codes with video-based explanations.

4.1 Study 1: Explaining Code

As previous work has shown, remote coordinators are adept at adapting their communication strategies for nearly any fidelity of shared information [17, 31]. In order to better understand how chat.codes affects programmers' ability to communicate, we conducted a qualitative evaluation with chat.codes and three alternative communication channels.

4.1.1 Setup. We recruited four experienced programmers and asked them to write and explain a piece of code for caching web requests. This task involved writing, modifying, and explaining code. In this study, we asked every participant to explain the same concept with four different communication channels:

- chat.codes through the its web interface;
- a synchronous chat and shared editor interface, analogous to [20–22, 27, 42];
- a synchronous video screencast with voice communication, analogous to [13, 29]; and
- an asynchronous forum posting, analogous to StackOverflow [36].

We randomized the ordering of conditions between participants and asked them to adapt their descriptions to be appropriate for whichever communication channel they were using. Before every stage of the study, we gave participants a five minute introduction to the communication tool they would use in that stage. For the chat.codes stage, this five minute introduction walked them through the process of creating code pointers and looking at previous versions of code. After every stage of the study, we asked participants to complete a short survey to evaluate the pros and cons of the communication tool. At the end of the study, we interviewed participants about their experience using each of the tools. Each study was performed in person and lasted approximately 1.5 hours.

4.1.2 Results. Usage Patterns: We observed two types of usage patterns between participants (but these differences were only apparent in the video and chat.codes conditions). Most (three of four) participants interleaved code edits with explanations in both cases. In chat.codes and in video, they made incremental code edits and explained their edits (in messages for the chat.codes condition and verbally in the video condition). Participants in the chat.codes condition did this interleaving frequently; every participant sent an average of 3.9 messages between every code edit. The other participant made more significant edits to their code that were not accompanied by explanations, and then went back to explain their edits step-by-step (in both the chat.codes and the video conditions).

We found that the ability to reference specific portions of code was important but that participants' methods for doing so varied by medium. In the chat.codes condition, every participant used code pointers to indicate regions of code (on average, they used one code reference every 2.5 messages). In the video condition, every participant highlighted code regions to indicate which region of code they were talking about. In the standard chat and forum conditions, participants typically referred to regions of code by description (e.g., "in the first `if` statement") or by referring to line numbers.

Reported Preferences and Comparisons: Two of the four participants reported that they preferred chat.codes the most (with video being the second most preferred). The other two participants reported that they preferred video the most (with chat.codes being the second most preferred). Every participant rated standard chat as third most preferred and forum postings as the least preferred. We did not observe any noticeable effect from task ordering on users' preferences.

chat.codes vs. video: For participants who rated the video condition highest, the most important feature of video was that it enabled communication through two mediums simultaneously by allowing them to write code and speak at the same time. However, for the other two participants, this aspect of video communication was actually a drawback. Neither was used to typing and talking at the same time and both found it difficult to get their timing right. One participant found that they frequently needed to backtrack in their video to correct something that they phrased incorrectly earlier. Chen et al. found a similar variation between users' preference for video vs. shared chat and noted that non-native speakers often preferred text chat over video [8] (all of our participants were native English speakers).

chat.codes vs. standard chat: All four participants independently identified code pointers and the ability to revert back to previous versions of code as being helpful and as the primary reason

they rated chat.codes above standard chat. As one participant said, code links "help me be sure that the other person is looking at the right thing". They also expressed concern with the fact that in standard chat, there was no easy way to "catch up" if they fell behind in an explanation. One participant also found chat.codes's diff tracking feature useful for themselves. As they were making code edits, they made a mistake and used chat.codes's diff tool to go back to a previous version of their code.

Further, participants reported a minimal learning curve for chat.codes. Explanations for code pointers and code history took under five minutes and none of our participants reported difficulty using either feature. Every participant in their post-task survey also agreed that most people would learn to use chat.codes very quickly.

chat.codes vs. forum posts: Every participant reported asynchronous forum posts to be their least preferred communication channel. Participants found it to be cumbersome to refer to regions of code and found it frustrating that forum posts did not provide an intuitive way to "point to anything". Thus, they found non-linear explanations (explanations that require jumping to various parts of the code) to be cumbersome in forum posts.

4.2 Study 2: Reading Explanations

We conducted a second study to better understand how students would interact with chat.codes's code pointers and code history.

4.2.1 Creating Equivalent Explanations. In order to understand how students would interact with code pointers and code history, we first wanted to create two equivalent explanations of the same concept: one that used these features (*"chat.codes"* explanations) and one that did not (*"standard"* explanations). In order to do this, we recruited an "instructor" participant and asked them to write two explanations for two pieces of example code (for a total of four explanations). The first piece of example code code was for the "Shannon Guessing Game", where a program tries to guess the next letter in a sequence of letters by storing bigrams from previous examples (for example, if the input is 'q', the output should likely be 'u'). The second piece of example code estimated the value of π using a Monte Carlo method (throwing "darts" at a rectangle and calculating how many land in a circle). The first example was 25 lines of Python; the second example was 15 lines of Python. Both required a combination of conceptual knowledge about the problem solving strategy and programming knowledge to fully understand the solution.

For both examples, we asked the instructor participant to write two explanations: (1) an explanation that uses chat.codes's code references and code history, using some of the usage techniques that instructors in Study 1 used and (2) another explanation as if they were writing an explanation for a discussion board that did not have code reference and history features. We asked the instructor to make both of these explanations as comparable as possible — to write explanations that were optimal for the given medium but to also ensure that both explanations had equivalent content. We then slightly modified this instructor's explanation by 1) adding pictures for the π approximation task in both explanations and 2) modifying some of the wording to ensure that both explanations were equivalent. Finally, we ensured that participants in the "standard" condition could find the correct code by adding any missing line numbers to the explanation (matching the convention that instructors used in study 1. In order reduce the "experimenter demand" bias where participants know which condition is the "treatment" condition, we presented both explanations (chat.codes and standard) within chat.codes (except the standard explanation did not use the code pointer or history features). Our four modified explanations (one with code versions and references, one without for each of the two code samples, π approximation and Shannon Guesser) were then used for the second part of the study, described below.

	Monte Carlo π				Shannon Guessing Game			
	Time (min)		Understanding		Time (min)		Understanding	
	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
standard	7.15	2.12	3.75	0.43	6.54	1.10	3.75	1.09
chat.codes	4.93	1.30	3.5	1.12	4.77	2.48	4.25	0.43

Table 1. The means and standard deviations for each task attempted and system condition used. Four people performed each task in the chat.codes condition and the other four in the standard (control) condition; each person switched conditions for the other task.

4.2.2 Study Setup. We recruited eight additional "student" participants (seven female, one male) from two university courses to evaluate the understandability of these explanations. All of these participants were students in an introductory Python programming course (3 participants were in a graduate Python course, 5 participants were in a different undergraduate Python course). Every participant had approximately 3 months of programming experience. Participants were given a short (approximately 2 minutes, self-guided) tutorial of the chat.codes interface, which they interacted with through their own personal computer. Each study was conducted in person.

We used a within-participants design — every student participant performed two tasks, one for the Monte Carlo π estimate and one for the Shannon Guessing Game. One explanation used chat.codes's code references and version history and one did not. We randomized which explanation used code references and counter-balanced the order in which tasks were done to account for learning effects and differences in task difficulty. After each explanation, we asked the participant to answer three questions that assessed whether they fully understood the code. We also asked them to fill out a survey with a self-assessment of their understanding of the explanation and the benefits and drawbacks of each explanation. After participants completed both sets of tasks, we then conducted a short interview where we clarified any points they raised in the surveys and we asked for additional feedback, including a comparison of which explanation style they preferred. Participants viewed both explanations within chat.codes, but we refer to the explanations that used code version history and references as the chat.codes condition in the experiment.

4.2.3 Results. All student participants correctly answered all the questions, so our analysis focuses on time to complete the tasks and students perceived understanding. Every student completed the task in the chat.codes condition faster, regardless of which task (Monte Carlo π or the Shannon Guessing Game) they had in that condition. For each student, we computed *time_diff* as the difference between the time the user spent doing the chat codes task and the time the user spent doing the other task. A summary of the means and standard deviations for each task type (Monte Carlo π or Shannon Guessing Game) by condition is shown for the outcomes of both time the user spent on the task and their reported understanding of the task on a 5 point Likert scale. The overall mean difference between the chat.codes and standard conditions was 1.99 minutes. The difference was significant in a paired t-test ($p < .01$). Because students took a little longer on average to complete the Shannon task, we also estimated a linear regression predicting time taken with task and condition as independent variables and fixed effects (dummy variables) for the individual users. Again, the 1.99 minute coefficient for the chat.codes condition was significant ($p < .01$); the coefficient for task was not significant. Thus, we can conclude with confidence that the chat.codes reduced overall task time.

Similarly, we modeled the understanding of the learner as self-reported on a 5 point Likert scale using a similar regression model. In this case, no significant difference was found.

When asked which explanation they preferred, all nine participants indicated that they preferred the "chat.codes" explanation, which made use of code references and code history. This is especially notable given that it was independent of task type and order and because participants used the chat.codes system for both tasks.

We also analyzed our collected data to find patterns in how participants used code pointers. Every participant except for one chose to inspect almost every code pointer as they went through the explanations, with some back-tracking. The participant who did not make use of these features, opting to instead read the explanation without code references or viewing intermediate pieces of code (this participant had a self-assessed understanding of 5/5), which highlights the flexibility students have in how they choose to read and understand explanations in chat.codes.

4.2.4 Participant Feedback. Every participant appreciated the ability to view intermediate versions of code in the chat.codes condition and made use of them. Every participant except one made use of the code references and history features, running intermediate versions of code an average of 18.1 times per task.

In interviews, nearly every participant expressed that their favorite feature of chat.codes was the ability to view intermediate versions of code (which many participants referred to as having "chunks" of explanation):

> P2 (on chat.codes): *"It was helpful to see the example built out ... to have it split up and be able to try the code out in different states of building."*
>
> P5 (comparing explanation styles): *"What stood out was [the chat.codes explanation] was chunked, had pictures, and connected explanation to code. The [standard explanation] was detailed about each line of code like comments, but [the] difference was it didn't highlight it and couldn't run it along the way, so it doesn't show you what the dictionary actually shows up....preference for the first one, even though it wasn't line by line; it was easier to understand"*

Participants also appreciated the code reference feature:

> P6 (on chat.codes): *"When you hover over things it will highlight them that was easier in terms of painting a picture"*
>
> P8 (on the standard explanation): *"At first [it] was confusing to follow to know which explanation was for which part of the code."*

4.3 Study 3: Comparison with Video Explanations

We conducted a third study to compare explanations in chat.codes with video explanations. Like chat.codes (but unlike regular chat and forum posts), recorded video sessions capture a full edit history. We recruited 48 participants through an online freelancing site. We selected for participants with at least some programming experience.

Every participant was randomly assigned to one of two conditions: *chat.codes* or *video* (a between subjects setup). Explanations in both conditions had the exact same content. In the chat.codes condition, that content was in textual form (with code pointers highlighting regions of code). In the video condition, that content was in audio/video form (with regions of code being highlighted in the video). The same series of code versions was used in both conditions. The explanation that participants looked was a tutorial on how to use the Turtle framework[2].

After reading the explanations, participants were then asked to write code that uses the Turtle framework to change pen colors and draw a shape. A correct solution was approximately ten lines of code. Each participants was given these either the chat.codes or video explanation as reference

[2]https://docs.python.org/3.3/library/turtle.html?highlight=turtle

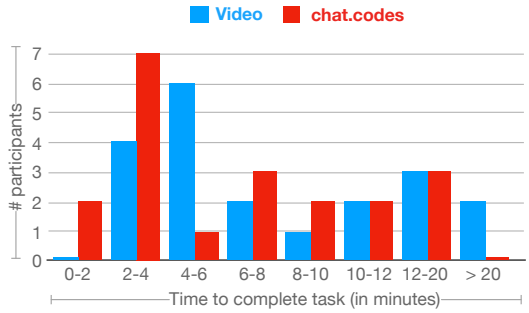

Fig. 6. The results of an evaluation of chat.codes and equivalent video tutorials with 40 participants who completed the task (study 3). chat.codes participants' results are shown in red (on the right) and video participants' results are shown in blue (on the left).

and was asked to perform a task with Turtle that required them to understand the explanations. Each study was performed remotely.

4.3.1 Results. Results are summarized in Figure 6. In each condition, four subjects failed to complete the task. 40 subjects completed the task and are included in our data analysis. For subjects who successfully completed the task, the outcome measure was total time to complete the task. In the chat.codes condition, the mean time was 7.18 minutes (sd 6.63). In the video condition, the mean time was 9.47 minutes (sd 5.24). However, the effects observed were not statistically significant at $\alpha \leq 0.05$ when applying an independent samples t-test ($p = 0.24$). Thus, we cannot conclude with confidence whether chat.codes leads to improved performance on average.

5 KEY FINDINGS FROM USER STUDIES

We summarize the key findings of all three evaluations:

- **Authors and readers both appreciated the ability to refer to regions of code**. For authors (instructors in our study), adding code references was easier than manually explaining which regions of code they were referring to. They also appreciated the fact that explicit code references ensured that anyone reading their description later on would be able to look at the right place. These references also allowed readers to feel confident that they were correctly interpreting an explanation.
- **Instructors can use code diffs as a self-explanatory description.** Instructor participants in our first study used chat.codes in several unexpected ways. One of the most noteworthy ways was by treating inline code diffs as self-explanatory descriptions, where an author would describe the high-level purpose of a change (why they are making it) and then make the change in code. For example, an instructor might write "now, we're going to make it work with negative numbers" and change the code appropriately. By capturing code edits, chat.codes makes it easy to associate the set of code changes with the high-level stated goal. This type of granular explanation could previously only be captured by VCSs (which are not conversational).
- **Tracking code edits made it easy for instructors to create instructive "intermediate" steps, which readers appreciated**. Every "student" participant in our second study reported liking the explanation that tracked code history because it grouped the explanation into steps. Participants could run code in intermediate steps to get a clearer picture of the final result.

- **Explanations in chat.codes are self-paced, giving users flexibility in how to read them**. One benefit of chat.codes is that it gives readers the ability to go at whatever pace is appropriate. For example, one chat.codes participant in study 2 chose to read message descriptions without using code pointers or intermediate code versions. This type of self-paced reading is difficult in video-based explanations.

6 IMPLEMENTATION

chat.codes is implemented in two parts: a server and a web-based client. The source code for our implementation of chat.codes is publicly available[3].

6.1 Server-Side Implementation

The server-side chat.codes code is a Node.js[4] web server. The server is responsible for tracking the state of any number of simultaneous, multi-user conversations. The server also tracks users — which users are involved in a given conversation and which users edited portions of code. In order to handle nearly simultaneous code edits from multiple remote users, chat.codes uses ShareDB[5], which uses Operational Transformations (OTs) [12]. OTs allows remote clients to avoid *merge conflicts*—conflicts that occur when two or more users edit the same region of code simultaneously—by managing and resolving these conflicts automatically. Several widely-deployed collaboration tools, including Google Docs, have successfully used Operational Transformations to enable real-time collaborations at scale. chat.codes stores code and chat history using a MongoDB[6] database.

6.2 Client-Side Implementation

The client-side chat.codes code is written in TypeScript using the AngularJS[7] web framework. chat.codes uses the web-baced Ace[8] as its code editor. Edits made in the code editor are translated into OTs, which are then sent to the server. The server merges these OTs with those from other users and sends updates to every client.

6.2.1 Computing Code Diffs. Every chat.codes client maintains a full history of code edits that occurred since a given conversation started. Instead of storing multiple complete copies of the code, chat.codes stores the *changes* to minimize the space that is used on the client. In other words chat.codes stores code by storing an initial version of the code: ($code_{start}$) and an ordered list of edits: ($edit_1$, $edit_2$, …, $edit_{latest}$). These edits are fine-grained, typically character-level insertions and deletions. For example, $edit_N$ might represent an insertion of the character 'x' at position 42 in the file-1.js. Every edit also stores a timestamp to allow chat.codes to accurately display chat messages and code edits in the order in which they happened.

The code diffs (like the one shown in Figure 5) are computed client-side. If a diff involves code versions $code_{diffStart}$ to $code_{diffEnd}$, the client code computes $code_{diffStart}$ by starting with $code_{start}$ and simulating edits $edit_1$ to $edit_{diffStart}$. It then computes $code_{diffEnd}$ by starting with $code_{diffStart}$ and simulating edits $edit_{diffStart+1}$ to $edit_{diffEnd}$. chat.codes then computes the line-level diff between $code_{diffStart}$ and $code_{diffEnd}$ using the Ratcliff-Obershelp diff algorithm [38].

Although coarse-grained diffs are displayed client-side, every client can still compute more finely-grained diffs. Future versions of chat.codes could take advantage of this to allow individuals

[3] https://github.com/(anonymized_for_submission)
[4] https://nodejs.org/
[5] https://github.com/share/sharedb
[6] https://www.mongodb.com/
[7] https://angularjs.org/
[8] https://ace.c9.io/

to customize the granularity of edits they want to be displayed. Despite the fact that the bulk of computations to compute diffs are done client-side in chat.codes, we did not see a noticeable performance degradation, even in longer (approximately 1 hour) discussion and code editing sessions.

6.2.2 Code Pointer Implementation. Every code pointer consists of three pieces of information: the *file* that the pointer refers to, the specific *region of code* within that file, and the *code version* that the pointer refers to. The file and region of code are explicitly stored as part of the code pointer message (e.g., `[this code](file3.js:L42,0-L44,3)`) although they are not displayed in the chat window. The code version is inferred from the timestamp of the message relative to the timestamps of code edits. When a user selects a code pointer (by either hovering or clicking it), chat.codes determines which region of code should be highlighted by applying every code edit before the code pointer's message (starting with $code_{start}$ and applying $edit_1$ to $edit_n$ where $edit_n$ is the last edit before the message). It then displays this code in the Ace code editor and highlights the relevant code region.

7 LIMITATIONS & FUTURE WORK

The evaluations described in this paper assess only one use case for chat.codes, instructor authored explanations intended to be consumed at a later time by learners. We intend to explore other use cases, involving a synchronous as well as asychronous use, two-way and multi-way conversation, peer to peer communication as well as expert to novice. We expect that additional design challenges will emerge as we conduct that exploration. Some of the anticipated challenges and potential approaches to resolving them are described below.

7.1 Encouraging Exploration and Experimentation

The most requested feature for chat.codes from participants in our second study was the ability to "branch" off from previous versions of code in order to experiment and run the code with different parameter values. This feature could be especially useful in courses that encourage students to build an understanding of code by "tinkering" with it. The major design challenges here will be keeping users oriented as they navigate through multiple branches of code revision, and helping users to develop a mental model of when their code window is shared versus individual.

7.2 References to Internal Program State

Although chat.codes allows users to point at regions of code, some explanations refer to aspects of a program's runtime state or output, as is possible with tools like Codechella [22]. We plan to explore ways to enable these kinds of references in future versions of chat.codes. The major design challenge is that up until now execution windows have been private, not shared; changing this will require providing users with a mental model of when execution state is shared and when it is not.

7.3 Conversation Search and Summary

Our evaluations indicate that chat.codes logs can be reused by other programmers. However, we do not know if conversations between users will be useful to later viewers in the same way that logs designed to serve as explanations are. Moreover, previous conversations can only be useful if users can find them. We plan to explore effective ways to allow programmers to find previous conversations that answer a given question by searching through previous codebases or messages.

7.4 Applications in MOOCs and Software Programming Teams

The growing demand for programming-related jobs has led to an increased demand for programming education [9]. Many institutions are finding it difficult to scale instructional resources to match the pace of rapidly growing enrollments in computer science courses [43]. Although Massive Open Online Courses (MOOCs) aim to provide better access to computer science education, they lack the type of personalized support that in-person courses provide [24]. However, it can be difficult to provide synchronous support at the scale of MOOCs or large traditional programming courses [6]. We plan to explore whether chat.codes can help instructors in MOOCs provide scalable personalized support by creating a searchable repository of chat.codes conversations for a given course.

We also believe that the features of chat.codes could be useful for programming teams — particularly distributed programming teams where project experts explain code to project newcomers. Like in the MOOC setting, these discussions could be useful for future project contributors. Thus, we also plan to explore how chat.codes could be useful in the context of programming teams and how it could integrate into their existing workflows and tools.

One difference between educational settings and programming teams is that in educational settings it may be desirable for students to find conversations that help reason about a problem without revealing a complete solution. We will explore ways to limit the provision of complete solutions, while still making chat.codes useful for question answering, debugging, and deepening student understanding.

7.5 System Features

We are also considering specific changes to the chat.codes interface based on feedback from our user studies. For example, we are considering adding features to help track targeted regions of code through edits, as Ginosar et al. have explored [18]. We are also investigating alternative ways to allow programmers to cluster sets of changes into groups. One possibility would be to allow programmers to explicitly specify a group of changes as being atomic. This convention might allow programmers to "point" at code deltas in a fashion similar to chat.codes's code pointers. We also plan to explore ways to allow programmers to manage permissions and code visibility (for example, so that observers can only read or edit limited portions of their codebase).

8 CONCLUSION

chat.codes is a communication tool for programmers that builds on state-of-the-art code communication tools by tightly coupling discussions with the code being discussed. By allowing programmers to add code pointers, to see the state of a code when any given message was sent, and to see summaries of code changes along with explanations, chat.codes enhances communication between programmers and allows conversations to be reused later on to improve both synchronous and asynchronous communication. In three evaluations, we found that chat.codes is effective for writing and for reading explanations about code. Instructors using chat.codes were adept at taking advantage of its features in writing explanations, and readers performed as well or better with chat.codes explanations as with text-only or video explanations.

One major hurdle in designing chat.codes was finding a way to embed deictic code references in human-readable chat logs in a way that they could be authored and edited either by typing text in the chat window or by clicking a mouse in the code window. It will be interesting to see how well this approach generalizes to other kinds of deictic references. For example, software tutorials might include references to particular screen elements; it is not clear whether our markdown approach would translate well to that situation.

We have opened up the design space for interfaces that use a linear text with embedded references to navigate through the history of some other artifact, in our case a code file. Other artifacts with multiple versions where this approach might be useful could include architectural blueprints and legislative bills. As described in the previous section, a major challenge still remains in using a linear text to navigate a branching history tree of artifacts rather than a single sequence of artifacts.

In the early days of CSCW, the Xerox Colab project articulated a fundamental tension around the degree of coupling between what multiple users experiences [39]. They articulated the What You See Is What I See (WYSIWIS) abstraction but also identified needs for individual work that would not be immediately visible to everyone. Among other things, they identified an issue of granularity of sharing that still echoes through the design of all collaborative tools. For example, in chat.codes and most chat systems, chat messages are shared only upon message completion, not a character a time. In most shared document editors, synchronization happens a character at a time and cursor movements are shared but mouse movements are not.

This fundamental design challenge of what is shared and not shared in a collaborative tool emerges in chat.codes with the code window. In the current design, only users looking at the latest version of the code share a code editor; when they look at a past code version, they temporarily exit WYSIWIS mode to see a private read-only view. Other rules for entering and leaving a shared editing mode are possible and deserve exploration, both in the context of shared code editing and collaboration activities with other artifacts that have version histories.

9 ACKNOWLEDGEMENTS

We thank all of our participants across all three studies and our reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No IIS 1755908.

REFERENCES

[1] Mark S Ackerman. 1998. Augmenting organizational memory: a field study of answer garden. *ACM Transactions on Information Systems (TOIS)* 16, 3 (1998), 203–224.

[2] Mark S Ackerman and Thomas W Malone. 1990. Answer Garden: A tool for growing organizational memory. In *COCS '90 Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems*, Vol. 11. ACM. Issue 2-3.

[3] Zane L Berge. 1999. Interaction in post-secondary web-based learning. *EDUCATIONAL TECHNOLOGY-SADDLE BROOK NJ-* 39 (1999), 5–11.

[4] Lori Breslow, David E Pritchard, Jennifer DeBoer, Glenda S Stump, Andrew D Ho, and Daniel T Seaton. 2013. Studying learning in the worldwide classroom: Research into edX's first MOOC. *Research & Practice in Assessment* 8 (2013).

[5] Peter Brusilovsky. 2001. WebEx: Learning from Examples in a Programming Course.. In *WebNet*, Vol. 1. 124–129.

[6] Lisa Chamberlin and Tracy Parish. 2011. MOOCs: Massive Open Online Courses or Massive and Often Obtuse Courses? *eLearn* 2011, 8, Article 1 (Aug. 2011). https://doi.org/10.1145/2016016.2016017

[7] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S Lasecki, and Steve Oney. 2017. Codeon: On-Demand Software Development Assistance. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM.

[8] Yan Chen, Steve Oney, and Walter Lasecki. 2016. Towards providing on-demand expert support for software developers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM.

[9] Parmit K Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, Brett Armstrong, and Philip J Guo. 2015. Perceptions of non-CS majors in intro programming: The rise of the conversational programmer. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 251–259.

[10] Soon Hau Chua, TONI-JAN KEITH MONSERRAT, Dongwook Yoon, Juho Kim, and Shengdong Zhao. 2017. Korero: Facilitating Complex Referencing of Visual Materials in Asynchronous Discussion Interface. *interface* 1 (2017), 6.

[11] Sarah D'Angelo and Andrew Begel. 2017. Improving communication between pair programmers using shared gaze awareness. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 6245–6290.

[12] Clarence A Ellis and Simon J Gibbs. 1989. Concurrency control in groupware systems. In *Acm Sigmod Record*, Vol. 18. ACM, 399–407.

[13] Floobits. 2015. https://floobits.com/ Accessed: September, 2017.

[14] Susan R Fussell and Robert M Krauss. 1992. Coordination of knowledge in communication: Effects of speakers' assumptions about what others know. *Journal of personality and Social Psychology* 62, 3 (1992), 378.
[15] Susan R Fussell, Robert E Kraut, and Jane Siegel. 2000. Coordination of communication: Effects of shared visual context on collaborative work. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM, 21–30.
[16] Susan R Fussell, Leslie D Setlock, Jie Yang, Jiazhi Ou, Elizabeth Mauer, and Adam DI Kramer. 2004. Gestures over video streams to support remote collaboration on physical tasks. *Human-Computer Interaction* 19, 3 (2004), 273–309.
[17] Darren Gergle, Robert E Kraut, and Susan R Fussell. 2013. Using visual information for grounding and awareness in collaborative tasks. *Human–Computer Interaction* 28, 1 (2013), 1–39.
[18] Shiry Ginosar, De Pombo, Luis Fernando, Maneesh Agrawala, and Bjorn Hartmann. 2013. Authoring multi-stage code examples with editable code histories. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 485–494.
[19] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 7.
[20] Max Goldman, Greg Little, and Robert C Miller. 2011. Real-time collaborative coding in a web IDE. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 155–164.
[21] Philip J Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 599–608.
[22] Philip J Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-User Program Visualizations for Real-Time Tutoring and Collaborative Learning. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE.
[23] Carl Gutwin and Saul Greenberg. 2002. A descriptive framework of workspace awareness for real-time groupware. *Computer Supported Cooperative Work (CSCW)* 11, 3-4 (2002), 411–446.
[24] Mark Guzdial. 2014. Limitations of MOOCs for Computing Education-Addressing our needs: MOOCs and technology to advance learning and learning research (Ubiquity symposium). *Ubiquity* 2014, July (2014), 1.
[25] Rebecca A Hines and Cynthia E Pearl. 2004. Increasing interaction in web-based instruction: Using synchronous chats and asynchronous discussions. *Rural special education Quarterly* 23, 2 (2004), 33.
[26] Jonathan Huang, Anirban Dasgupta, Arpita Ghosh, Jane Manning, and Marc Sanders. 2014. Superposter behavior in MOOC forums. In *Proceedings of the first ACM conference on Learning@ scale conference*. ACM, 117–126.
[27] Cloud9 IDE Inc. 2010. Cloud9 IDE. https://c9.io Accessed: September, 2017.
[28] GitHub Inc. 2014. Atom Editor. https://atom.io/ Accessed: September, 2017.
[29] HackHands Inc. 2015. Hack.hands(). https://hackhands.com/ Accessed: September, 2017.
[30] David H Jonassen. 2004. *Handbook of research on educational communications and technology*. Taylor & Francis.
[31] Robert E Kraut, Darren Gergle, and Susan R Fussell. 2002. The use of visual information in shared visual spaces: Informing the development of virtual co-presence. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*. ACM, 31–40.
[32] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2857–2866.
[33] J Manning and M Sanders. 2013. How widely used are MOOC forums? A first look. *Signals: Thoughts on Online Learning* (2013).
[34] Markdown. 2004. https://daringfireball.net/projects/markdown/ Accessed: September, 2017.
[35] Robert McGuire. 2013. Building a sense of community in MOOCs. *Campus Technology* 26, 12 (2013), 31–33.
[36] Stack Overflow. 2015. Stack Overflow. https://stackoverflow.com/ Accessed: September, 2017.
[37] Murat Oztok and Clare Brett. 2011. Social presence and online learning: A review of research. *International Journal of E-Learning & Distance Education* 25, 3 (2011).
[38] John W Ratcliff and David E Metzener. 1988. Pattern-matching-the gestalt approach. *Dr Dobbs Journal* 13, 7 (1988), 46.
[39] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. 1987. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. *ACM Trans. Inf. Syst.* 5, 2 (April 1987), 147–167. https://doi.org/10.1145/27636.28056
[40] Michael Tsang, George W Fitzmaurice, Gordon Kurtenbach, Azam Khan, and Bill Buxton. 2002. Boom chameleon: simultaneous capture of 3D viewpoint, voice and gesture annotations on a spatially-aware display. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*. ACM, 111–120.
[41] Chih-Hsiung Tu and Michael Corry. 2003. Designs, management tactics, and strategies in asynchronous learning discussions. *Quarterly Review of Distance Education* 4, 3 (2003), 303–15.
[42] Jeremy Warner and Philip J Guo. 2017. CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 1136–1141.

[43] Chris Wilcox. 2015. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 90–95.

[44] Dongwook Yoon, Nicholas Chen, Bernie Randles, Amy Cheatle, Corinna E Löckenhoff, Steven J Jackson, Abigail Sellen, and François Guimbretière. 2016. RichReview++: Deployment of a Collaborative Multi-modal Annotation System for Instructor Feedback and Peer Discussion. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. ACM, 195–205.

[45] YoungSeok Yoon and Brad A Myers. 2015. Semantic zooming of code change history. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 95–99.

[46] Young Seok Yoon and Brad A Myers. 2015. Supporting selective undo in a code editor. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 223–233.

[47] Sacha Zyto, David Karger, Mark Ackerman, and Sanjoy Mahajan. 2012. Successful classroom deployment of a social document annotation system. In *Proceedings of the sigchi conference on human factors in computing systems*. ACM, 1883–1892.

Received February 2007; revised March 2009; accepted June 2009