AN ABSTRACT OF THE DISSERTATION OF

Seyed Soroush Ghorashi for the degree of Doctor of Philosophy in Computer Science presented on June 9, 2017

Title: Evaluating the Impact of Live Programming on Collaborative Software Development.

Abstract approved:

_____

Carlos Jensen

Collaboration is tricky, but often beneficial in the context of numerous software related activities, from learning core concepts, to the design and implementation of large software products. The growth of online classes, from small structured seminars to massive open online courses (MOOCs), and the isolation and impoverished learning experience some students report in these, points to an urgent need for tools that support remote pair programming in a distributed educational setting. In "the real world" software developers and designers work together to solve common problems, and meaningful and effective designer-developer collaboration improves the user experience. Supporting these with today's often distributed work model presents important challenges.

Two key techniques which are believed to be effective in promoting better coordination and collaboration are collaborative coding and live programming. Collaborative coding allows all the team members to get involved in the development process, and live programming enables them to see what they are building effortlessly and in real time.

In this work, we first describe Jimbo, an integrated development environment (IDE) based on collaborative and live programming techniques, and a set of user studies aimed at evaluating whether these techniques are effective in promoting better coordination and collaboration in two different settings; distance learning and design-focused

software development. Our results show that these techniques can improve the learning experience through pair programming and a tight code-artifact feedback loop. We will show how collaborative coding and live programming can help designers and developers bridge their knowledge and language gaps and develop mutual understanding, allowing designers to join the development process as first-class citizens – not dependent on the coders to compile and share output – or being forced to become coders.

Evaluating the Impact of Live Programming on Collaborative Software Development

by
Seyed Soroush Ghorashi

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 9, 2017
Commencement June 2017

Doctor of Philosophy dissertation of Seyed Soroush Ghorashi presented on June 9, 2017

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Seyed Soroush Ghorashi, Author

ACKNOWLEDGEMENTS

My thanks go to Dr. Carlos Jensen, my major adviser, for providing valuable feedback and encouragement throughout my years at Oregon State University. Thanks to my current and past committee, Dr. Ronald Metoyer, Dr. Cindy Grimm, Dr. Alex Groce, Dr. Margaret Burnett, and Dr. Christopher Scaffidi for providing many insights into this research. A special thanks to my former colleague, MingChieh Chang, who helped with the design and development of the first version of the prototype I used in this research. Thanks to the rest of the Human-Computer Interaction group for listening to countless practice talks, participating in pilot studies patiently and helping me conduct my user studies: Jennifer, Iftekhar, Kalin, Dedrie, Amir, Rahul, and Chad. Thanks to all my friends in Corvallis with whom life was still enjoyable and fun while away from home: Nourieh, MJ, Hossein, Meisam, Abbas, Chadwick, Alireza and Vahid.

Finally, and most importantly, sincere and heartfelt thanks to my wife Nahid, my parents Abolhassan, Sorour, my sister Sally, my brother Soheil, my brother in law Hossein and my lovely nephew Ryan, for their endless encouragement throughout my education. Their unconditional love and support made my journey far from home possible, I feel very lucky to have such an amazing and supportive family behind me.

CONTRIBUTION OF AUTHORS

The following authors contributed to chapter 2: Carlos Jensen

The following authors contributed to chapter 3: Carlos Jensen

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

1 Introduction

One of the most important aspects of human nature is the drive to communicate and collaborate with each other. Regardless of the activity that we are involved in, it's through collaboration that we often achieve our goals and satisfy our needs. In the context of computer science, collaboration is tricky, but often beneficial for a variety of software related activities, from learning core concepts, to the design and implementation of large software products. In this work, our focus is on how we can improve the collaboration between users working together in two different yet related settings: Distance Learning and Design-centered Software Development.

1.1 Distance Learning Settings

One of the most popular and effective collaboration methods used in CS education is pair programming, which has been shown to be a very beneficial technique for teaching and engaging students with programming and new computing topics. The need for tools that support remote pair programming is becoming pressing with the growing popularity of massive open online courses (MOOC). While employing pair programming in a collocated classroom setting is relatively straightforward, there is a dearth of good options for distributed classroom settings. As students struggle to master concepts and build confidence in their skills, a tight code-artifact feedback loop/mechanism that allows students to verify that a change had the intended result is important.

In the second chapter, we start with a review of related work, explaining the design requirements that needed for a tool to support collaborative learning. Then we describe our tool, Jimbo – Educational Edition, and explain how it addresses various issues students face in remote pair programming. Next, we explain the user studies that we conducted to evaluate our tool and then present the results. We conclude with a discussion of challenges to developing collaborative tools to be used in classroom setting.

1.2 Designed-centered Software Development

The development of software systems is a collaborative process, where team members work together to solve a problem by producing quality code. The designer-developer relationship at the heart of many of these collaborations is the force that moves a software project toward success. Unfortunately, in current software development practices, designers have no direct engagement with developers in the development process, although the products performance depends on both. If we want to improve this relationship, and encourage better software products, we need to build development tools that improve the collaboration and work-flow for designers and developers.

In chapter 3, first we explain how similar design goals as in chapter 2 can improve the relationship between developer and designers. Then, we describe our tool, Jimbo – Professional Edition, which is an upgrade to our educational edition with a different UI, optimized for collaboration/coordination in professional software development. Finally, we report the results of a user study that shows how collaborative coding and live programming can help designers and developers bridge their knowledge and language gaps and develop mutual understanding, allowing designers to join the development process as first-class citizens – not dependent on the coders to compile and share output – or being forced to become coders.

1.3 Contributions

Contributions of this research includes:

- Thorough survey on existing research about collaborative learning, methods and tools developed to support pair programming (both traditional and remote style) and live coding in education (both collocated and distance settings).
- Design and development of a novel IDE called Jimbo – Educational Edition; a development tool that offers both remote pair programming and live coding for teaching introductory web development courses to novice students.
- Conducting multiple user studies in both collocated and distance learning settings to evaluate the effects of remote pair programming and live coding in improving the active teaching/learning experience.

- Thorough survey on existing research about collaborative software development and live programming, techniques, practices and tools designed to support collaboration for software teams.
- Design and development of a novel IDE called Jimbo – Professional Edition; a web-based collaborative IDE with live preview, optimized for collaboration between web developers and designers.
- Conducting a user study to evaluate the effects of collaborative coding and live programming in improving the relationship between the developers and designers.

2 Supporting Learners in Online Courses Through Pair Programming and Live Coding

Soroush Ghorashi, Carlos Jensen

School of EECS
Oregon State University
Corvallis, Oregon, 97331, USA
{ghorashs, carlos.jensen}@ oregonstate.edu

Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA

2.1 Abstract

Pair programming has been shown to be a beneficial and popular technique for engaging students and improving learning outcomes in programming and related classes. While using pair programming in a collocated classroom setting is relatively straightforward, there is a strong lack of good tools and options for distributed classroom settings. The growth of such classes, from small structured seminars to massive open online courses (MOOCs), and the isolation and impoverished learning experience some students report in these, points to an urgent need for tools that support remote pair programming in a distributed educational setting. This paper explores the requirements and needs of online learners in Computer Science through a literature survey. To validate these requirements, we implement a collaborative development environment aimed at improving the learning experience through pair programming and a tight code-artifact feedback loop. We conclude by presenting the results of a user study in both collocated and distributed classroom settings.

2.2 Introduction

Collaborative learning is a social process in which students learn by working with others. It is often used in conjunction with other methods aimed to make the learning an active process in the classroom, particularly appropriate for computer science (CS) due to the ever-increasing complexity and change [39]. The development of software is usually a highly collaborative process, where programmers, designers, managers, and other stakeholders work together to solve a problem through code. Vessey and Sravanapudi [55] showed that about 70% of a typical software engineer's time is spent on collaborative activities. Thus, the use of collaborative learning in teaching computer science is consistent with current practices in software development.

Pair programming is one of the more effective collaborative learning techniques, where two students work to solve a common problem on the same computer. One programmer is the "driver," which means that he controls the keyboard and mouse, while the

"navigator" reviews the code that the driver is writing. Research has shown that pair programming technique is very effective for teaching programming courses [2, 9, 36].

The importance of collaboration in CS education has been confirmed by numerous studies in both lab and classroom settings. Academic achievement is enhanced when students learn with others compared to when they learn alone [25, 41, 50]. Williams et al. also found that not only are students who work collaboratively on programming tasks 40-50% faster than stand-alone learners, they also write software with fewer defects, and the practice has a positive impact on the confidence of the students [60, 61]. McDowell et al. found that students who use pair programming not only produce better code but also perform significantly better on final exams compared to those programming alone [36].

Pair programming requires two programmers to work together on the same computer, but the trend toward geographically distributed teams make long-distance adaptations necessary. A modified model, sometimes called remote pair programming (RPP), adopts looser roles than traditional pair programming. Developers swap roles (driver/navigator) more frequently, and without coordination. Here, novice programmers can learn by working as part of a programming pair, contributing code as well as observing others and performing code reviews, without necessarily being part of an explicit role-pair. Collaboration is instead done more informally through IRC or other communication channels. It has already been shown that RPP has the same benefits of the collocated pair programming mentioning better communication and cooperation within the group [3].

While pair programming is relatively easy to implement in a collocated class, it is a social protocol after all, it is more challenging for online classes due to lack of tools and infrastructure. In most classrooms, students easily collaborate with each other and their instructor. In an online setting, the mechanisms for interaction are limited – either due to the remote nature of the class, or because the scale of programs such as the more popular MOOCs – which can have thousands of students per session across different

time zones, make it difficult for an instructor or students to coordinate. Thus, while online courses improve the availability of education, the learning experience is sometimes impoverished compared to collocated classes. Design and implementation of tools that support remote pair programming has its own challenges.

Another barrier to collaboration, or even hands-on learning in online/distributed settings is setting up and troubleshooting tools and development/application environments with limited support. This can be an insurmountable hurdle for students, and often leads instructors to settle for a lowest-common-denominator approach to technology adoption in the class. This is a problem that should not be ignored, as it can even consume an inordinate amount of time in collocated classrooms. To better support learners in online classes we need better tools.

To better support learning in online CS classes we need tools that allow instructors to easily track students' progress and intervene to address issues as they emerge rather than wait for students to fail. Because of the scale of some of these classes, and the diversity of contexts and technology available to students, a 0-set-up learning environment would be desirable. This means an environment that works reliably on every modern personal machine, brought to the students through for instance a web-browser. This allows students and instructors to focus on the material at hand rather than the supporting technology.

The rest of this document is organized as follows: We start with a review of related work, explaining the design requirements that needed for a tool to support collaborative learning. Then we describe our tool, and explain how it addresses various issues students face in remote pair programming. Next, we explain the user studies that we conducted to evaluate our tool and then present the results. We conclude with a discussion of challenges to developing collaborative tools to be used in classroom setting.

2.3 Related Work

Online classes have grown in popularity, and have come to occupy an important role in secondary education, and especially in CS education, where students and instructors are

perhaps more comfortable with the required technologies. Within online classes there is of course great diversity in how these are organized, what technology is used to teach them, whom they target, and how they fit into a greater educational context.

On one end of the spectrum, in terms of size and popularity, MOOCs deliver learning content online often at no cost and no limit on attendance. In this model, course materials are presented using recorded lectures, problem sets, readings and quizzes. However most of these providers also offer interactive ways of communication between students and professors, such as discussion forums. MOOCs initially gained prominence in the summer of 2011 when 160,000 people from around the world enrolled in a free online artificial-intelligence course offered by Stanford University, with 23,000 completing it [56].

The MOOC model is not the only online education model. On the other end of the scale, many universities are looking at using online classes as a way of offering courses which otherwise would not see sufficient local demand, or which cannot be fit into a regular teaching schedule. These classes are often taught with an instructor and TAs, at regular intervals, in much the same way as regular university classes, but with the advantage of increased flexibility and reach. One such program is run by <major US university>, through their e-campus program, open to both online and on-campus students. Over the last two years, enrollment in such CS classes has gone from one to two hundred students to over two thousand.

While on-campus students are easily able to interact with their professors and teaching assistants to resolve problems, for MOOC students the opportunities for these kinds of interactions are more limited, which may limit academic success and learning in these courses. Wouters et al. [62] suggest that learning improves through collaborative learning and instructional support. Warren et al. [57] suggest that the current major problem with MOOCs is that they are unable to offer the social experience of on campus courses.

Researchers have studied why collaborative programming is more effective than individual programming in an education setting. Williams et al. found that peer-pressure helps students concentrate and learn, which helps them to do better compared to the students who work individually [61]. Werner et al. suggest using pair programming to engage women in computer science, as this approach tends to lower social barriers that often limits their participation [58]. Collaborative learning also teaches students leadership, coordination and conflict management, essential for success in their future professional lives [45].

Another popular practice is "live coding," where an instructor writes code in front of students, exposing their thought process. While this is more effective than just showing the final solution to students, it is a passive learning technique. Gaspar and Langevin [17] successfully used a student-led version of this approach to engage students in active learning and expose their thought process to the instructor for more in-depth feedback. Here, the thought process is what matters, not the final product.

To better support student-led live coding and engage all students during a class session, having a tight code-artifact feedback loop that allows students to verify that a change had the intended result is important. Current software development workflows require developers to write code "blindly," only determining if their changes had the desired effect after they compile and run their program. If a learner wants to further change the program, they have to go back to the code, edit, compile and run the program again. This process is even more painful when working with partners, or on a UI-heavy application.

In the next section, we discuss key features that educational tools must incorporate in order to support classes that follow this collaborative learning model in either collocated or online settings.

2.4 Supporting Collaborative Learning

To better understand current practices that support pair programming and learning in both collocated and remote settings, we did a survey of tools described in the literature.

We started our search with the following conferences: CSCW, CHI, SIGCSE and keywords: pair programming, remote pair programming, collaborative learning, active learning, live coding/live programming, etc. After filtering for false positives, this identified more than 50 papers. From there we refined our list by looking up cited works. From this literature review we identify a set of key practices that should improve the learning experience of novice CS students in both collocated and remote classroom settings. These include on the social side:

2.4.1 Support both synchronous and asynchronous collaboration with remote students and instructors.

Supporting both synchronous and asynchronous collaboration is important, especially in the case of distributed classrooms, where students and instructors can be in different time-zones, and often combine classes with other commitments. However, asynchronous collaboration is also important in the case of collocated learning, as it captures more context.

A major characteristic of tools that allow synchronous coding is that they follow the WYSIWIS (What You See Is What I See) [51] metaphor, meaning that all users see the exact same document. GROVE [15], ShrEdit [37], DistEdit [29] and Flesce [8] are all examples. One of their key challenges is making sure the code is always error-free, as any coder may compile and run the code at any time, regardless of what the others are doing. Research tools such as Collabode [18] have been developed to address this issue, only sharing modifications that result in an error-free code, acting as an automatic safety buffer.

In terms of asynchronous tools, these often build on a shared code repository and mechanisms for automatic merging and conflict resolution. Subversion, Git and CVS [3] allow multiple coders to check out the same file as long as any changes are later synchronized in the repository. These tools allow coders greater freedom, but require the team to work harder to coordinate, and can lead to problems when work has to be merged, especially if developers delay such actions [5].

Several tools have been developed to provide real-time awareness of code changes in order to facilitate coordination and identify conflicts early. FASTDash [4] and ProjectWatcher [49] provide visualizations of data directly gathered from developers' workspace. Palantír [1, 47, 48] shows who is modifying what part of the code and alerts developers to emerging conflicts. Syde [21] follows the same approach, but reduces false positives through abstract syntax tree (AST) analysis. Crystal [5] proactively watches the code and precisely identifies and reports conflicts.

2.4.2 Integrate communication features, including text and audio to support collaboration and enquiry.

The integration of communication features into the IDE could help students discuss and learn from each other without losing focus on the code. One key factor to team success is the flow of communication between team members [6, 28]. The most common type of communications in coding teams is the face-to-face meeting. However, as classes get geographically distributed in remote learning systems, collaboration gets more challenging [14, 23]. Researchers have tried to overcome this by mimicking physical meetings in video conferencing systems [22, 25, 33], or more lightweight systems such as email, instant messaging [24], or even within the source control system itself [13]. Of these, email is by far the dominant mechanism due to its low learning curve and flexibility. Most of these communication tools (Skype, IRC, email, etc.) are not directly integrated with the IDE, which can lead to a disconnect between code and discussion, or simply wasted effort by frequent context switching.

While it is important to facilitate both synchronous and asynchronous collaboration – providing the most flexibility for students, synchronous collaboration is perhaps the most important as it makes it possible for students to engage in RPP at any time and from any location without worrying about syntactic consistency. A text-based chat system is a must, as coders share code snippets and links to resources. An audio chat system provides virtual presence and makes it easier to coordinate and collaborate.

2.4.3 Support awareness of other's activities in order to facilitate collaboration and remote pair programming.

Another fundamental requirement to supporting collaboration is awareness. Dourish and Belloti define awareness as "an understanding of the activities of others that provides a context for your own activity" [12]. Awareness is required to coordinate team activities, but can be distracting if it interrupts or requires too much attention from developers. It is not an easy thing to address in coding, as we juggle the need for asynchronous editing for some developers, and the need for real-time preview of the resulting code for others.

2.4.4 Integrate live preview to close the code-artifact feedback loop and support student-led live coding practices.

Live programming is a technique where programmers can re-execute a program continuously while editing [19]. Some recent live programming systems include Superglue, Flogo II and Lively Wiki [20, 30, 34]. Khan academy recently deployed a version of a live programming environment in an online course for students with no programming experience successfully [44]. While live programming shows users a copy of the final output, it does not provide a mapping between the output and the code, as done by step-based debugging or tracing with print statements. Researchers are trying to enhance live programming environments by focusing on debugging [35].

Live preview is a live programming technique that shows the code output immediately upon any changes to the code and it best fits UI-heavy application development such as websites. One advantage of combining a live preview component with pair programming is that live preview supports a distributed and thus scalable way of engaging in live coding. This would allow all students to benefit from this practice during short class periods, whereas before only a handful of them would get the chance to practice this technique.

On the code side, the key practices we identified include the following features, which though not necessarily unique to the needs of students in online CS course nonetheless simplify the process of programming and learning:

- A simple, web-based zero setup programming environment, which is easy to use by novices. This includes support for code completion and other advanced tools.
- Supporting code rewinding and automatic back-ups to minimize the cost or fear of mistakes and experimentation.
- A feature-rich code editor with state of the art support tools and scaffolding.

At the heart of any popular and successful IDE sits the code editor. Modern code editors include several important features designed to scaffold the task of programming. This includes features such as syntactical highlighting, function completion and inline documentation, automatic indentation and even auto-correction and spell checking. All of these features are aimed at helping the student focus on the logic of the program rather than the minutia of syntax.

Many of the features described above have been implemented in previous tools but never in a single IDE (most notably Plantír [48], Syde [21], CoRED [31], Eclipse JAZZ [8], Collabode [18] and Brackets [5]), and their efficacy in supporting learners has never been determined. It is conceivable that features which in theory sound helpful, or in isolation work, end up working against each other or not actually supporting the needs of learners. In order to determine whether this set of design considerations and requirements really help learners, we implemented a new IDE, called Jimbo, and performed an evaluation in both a collocated and remote learning setting.

2.5 Jimbo, A Collaborative Development Environment with Live Preview

In order to determine whether the design requirements identified above really had a positive effect on learners, we implemented an IDE which included all of them. Jimbo is an IDE for HTML5 development that enables students to more easily collaborate on a project. We have tried to make the user interface easy to learn and memorable, but have also considered external consistency with other popular IDEs. Jimbo is a web-based IDE, which means that students only need a standard web browser and there is no setup. This is important, especially in a remote classroom setting. Next we describe the

core functionality and design decisions made in Jimbo. Figure 2.1 gives an overview of our system architecture.

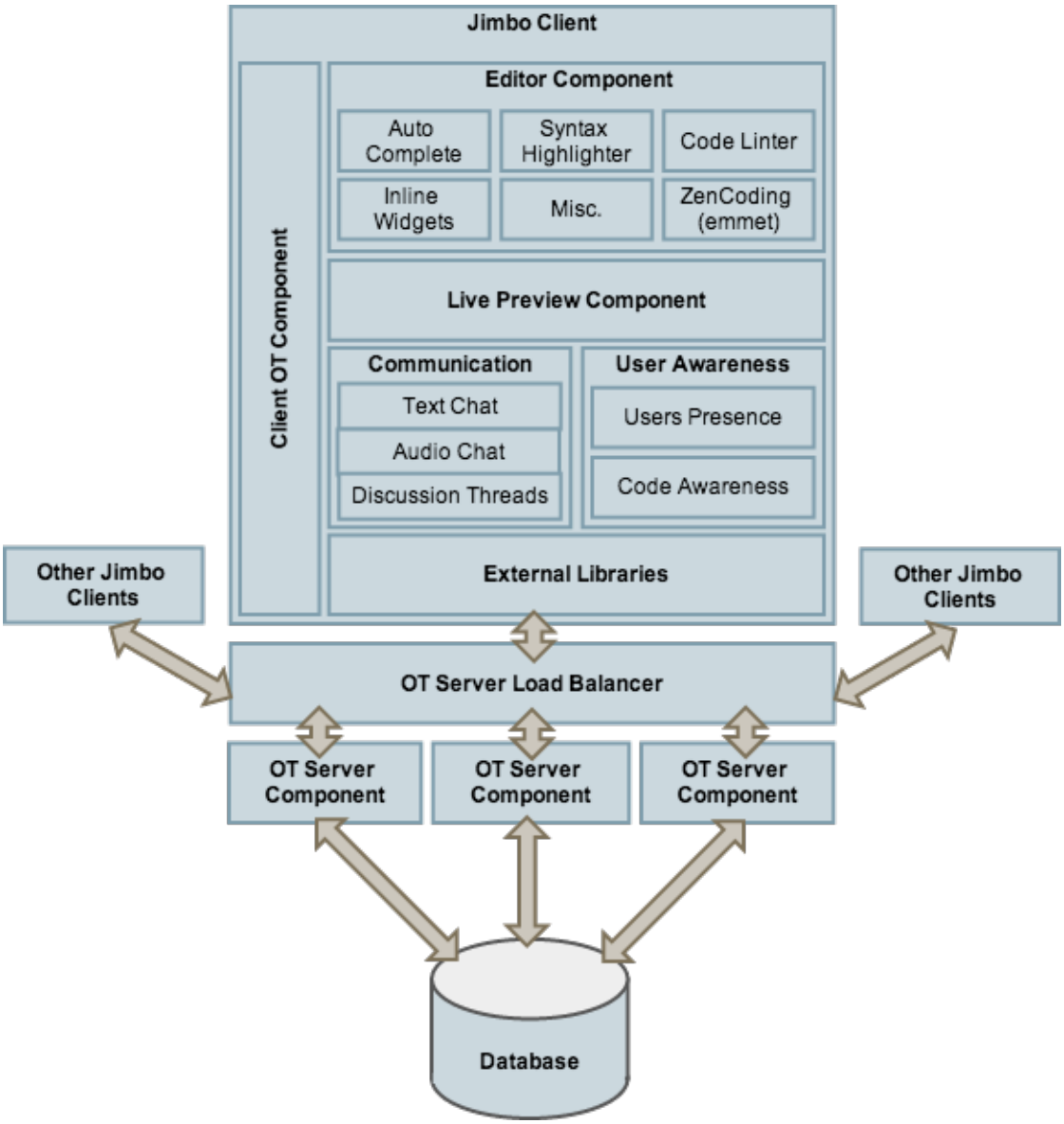

Figure 2.1 Jimbo's system architecture.

2.5.1 Main View

In Jimbo, any student can define a new project or go to an existing project linked to their account, with each project having a unique URL. Each project consists of four different stacks of “pages” (files): one for html code, one for JavaScript code, one for CSS and

one for JSON data. These are represented as a pile of paper, and the editor is overlaid on top (Figure 2.2d). These editors are easily accessible through a standard tab view. Jimbo's code editor also provides a set of features commonly found in modern IDEs, including code auto-completion, syntax highlighting, find/replace, linter, etc.

The left side of the screen is dedicated to the live preview. (Figure 2.2g) A list of currently online users is on the top of the screen with avatars and names (Figure 2.2a). Jimbo also has helper widgets for numerical and color values, which allows users to modify values using sliders and pickers instead of typing. This allows users to modify their code and quickly see the effects of their modifications in the live preview panel.

2.5.2 Synchronous Collaboration

The most important feature of Jimbo is synchronous collaboration. The number of defects in code tends to rise with the amount of parallel work [43] and developers sometimes avoid this kind of development to avoid having to resolve conflicts [19]. To minimize problems, Jimbo acts such as a real-time Git tool using an Operational Transformation (OT) algorithm [52].

OT is a technique that provides eventual consistency between multiple users working on the same artifact without retries, errors, or data being overwritten using simple insert/delete operations. In the basic form, the server keeps a state space per connected client, which can be memory-intensive and can make the transformation algorithms complicated. In Jimbo however, we made the process simpler and more efficient by requiring the server to acknowledge clients' operations before they can send new ones. This means that any client can at most have one un-acknowledged operation in flight. The client OT stores other users' operations in the localStorage of the browser, only sending the next when the last in-flight operation has been completed. Thus the server only needs to keep one state space for all connected clients.

The server also keeps different snapshots of each code file. If multiple collaborators try to edit the same code at the same time, one of the edits will be received and applied first and then the server transforms and applies the other edits using the state space. OT

algorithm makes sure that the commits and updates happen automatically and the code file is consistent for everyone no matter the order in which operations are applied to the shared file. This enables users to work offline and sync their edits with the server later; the OT algorithm takes care of the conflicts.

2.5.3 Code Rewind

This feature enables students to see how others are contributing to the project, maintain situational awareness, and identify whom they need to communicate with based on code ownership. The content is color coded to show who edited what part of the code. This also allows students to learn by example, and solves the concern of students getting a "free ride" in the class by allowing an instructor to see everyone's contribution to the team. The system also tracks lines of code being copy/pasted, should this be necessary in case of plagiarism. Another use of this feature is to determine the provenance of bugs introduced into the code.

2.5.4 Communication

Jimbo explores novel ways of integrating both synchronous and asynchronous communication. Jimbo implements discussion threads, a semi-synchronous communication method. These threads are associated with specific lines of code as inline comments. This allows developers to add and preserve contextual and design information, often generated in discussions with collaborators. Instant messaging/discussion system can be accessed through the chat icon on the right side of the preview section. Here users can chat or join a video/audio session (Figure 2.2i).

2.5.5 User Awareness

The main purpose of an awareness system is to help coordinate tasks. We follow the "continuous coordination" model introduced by van der Hoek et al. [54]. The primary responsibility of such a system is to notify students of events relevant to them, such as code changes, comments to discussion threads, user presence, etc.

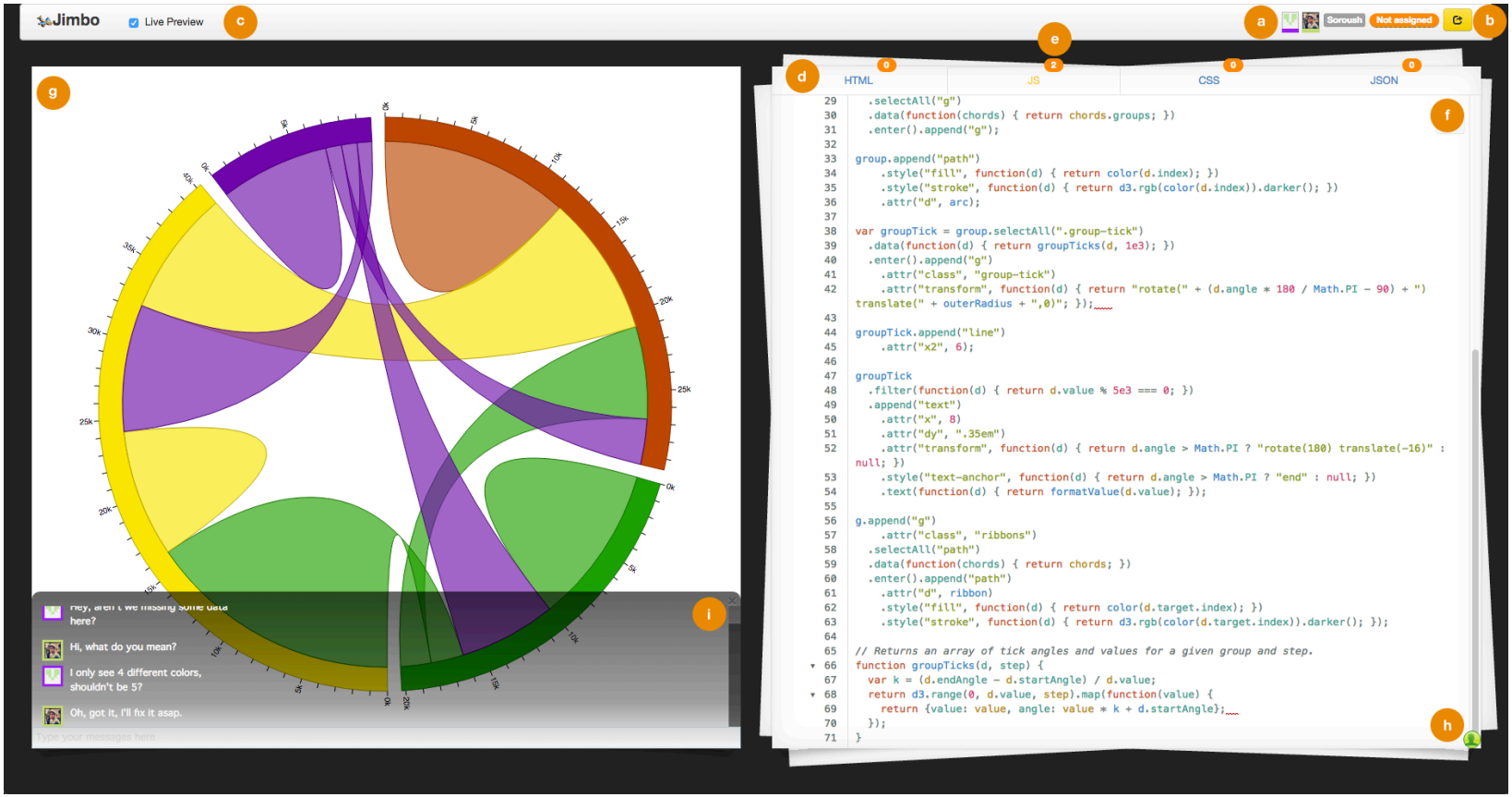

Figure 2.2 (a) Online developers' avatars with their names as tooltip. (b) Easy share button. (c) Live preview toggle (d) Code editor area. (e) Number of active developers in each editor, here both developers are active in the JavaScript editor. (f) Editor viewport. (g) Real-time preview panel. (h) Code change notification for the case that developers are in the same project but different viewports, here the change is happening somewhere after line 71. (i) Instant messaging panel.

Jimbo has a channel based notification system [42, 53] using push notifications [7, 16]. The editors' tab contains information about the developers currently working within each view. If developers are in the same editor, they can see each other's cursors.

Developers are notified about changes in the code that lead to a change in the preview panel in one of two ways:

- Developers in the same editor: It highlights the line changing and the student making the change.
- Developers in different editors: Jimbo highlights the editor name to let others know what editor the change is coming from.

2.5.6 Live Preview

This feature provides an immediate connection between the code and the output so students get feedback for changes to their code. This leads to fewer iterations, more immediate feedback, which means faster coding and better learning. It also streamlines collaboration between students in the same team. Using this feature with the communication features already discussed, they can provide immediate feedback as code is edited, in the pair programming tradition.

This feature only allows safe and error free code to be run in the live preview. In the case of buggy code, the live preview panel will show the last error-free code. Students can turn off the live preview feature to mitigate distractions at will. This is done to accommodate different learning styles.

2.7 Evaluation

We sought to evaluate Jimbo's effectiveness in improving learning in both collocated and distributed classrooms. We chose to do this through two between-group user studies, one in a collocated (classroom) setting and one in a distributed setting (online class). Both user studies were conducted at Oregon State University, using Computer Science students as subjects. Participation was voluntary, and all participants gave informed consent prior to participation.

The focus of our evaluation was not to determine whether pair programming is better than individual efforts. We wanted to determine whether we could streamline the use of pair programming and live coding in a way that would lead to an improved learning experience. Therefore, we structured our control groups as students working on assignments individually (as they normally would in these classes), and our experimental groups as pair programming groups. The measure of success would therefore not be whether students who worked together did better (which we took as given), but rather whether students could collaborate in a low-effort enough way to make the experience worthwhile and attractive.

The reason for studying both an online and a collocated class was to better evaluate our design goals. While online classes suffer from fewer opportunities for social interaction and collaboration between students, and between students and their instructors, on campus classes are less flexible in terms of time, and students have to be on task as much as possible.

Both studies followed the same format: First participants in the experimental condition received a short tutorial on how to use our tool. Next, both the control and experimental groups were introduced to a new programming concept from the course curriculum and asked to complete an in-class programming assignment designed by the course instructor. The experiment took place in week 7 of 10 of the term to ensure students were already familiar with the basics. In the control group, students could use any tool they chose, and the experimental groups used Jimbo. In most cases students in the control group used tools they had already installed and knew how to use.

2.7.1 Collocated User Study

The course we chose for this study was a small (16 students) graduate-level data visualization class. Here students learn how to visualize different types of data using d3.js library and other web technologies. This class requires students to master several new concepts over a short period of time (10 weeks).

Setup and participants

All 16 students enrolled in the class agreed to participate in our study. Subjects were all graduate students (4 female) with an undergraduate computer science background. We randomly assigned participants into two groups of 8; A control group and an experimental group. All but one participant had previous experience with pair programming, but only one participant had used a real-time collaborative tool before. We did not coach subjects on pair programming. Our control group worked on the programming task as individuals, using their own favorite code editor in the classroom with their instructor present (their preference).

Table 2.1 Collocated User Study.

	Years of Programming experience	Years of Web Dev experience	Self-rated HTML proficiency (10=expert)	Self-rated CSS proficiency (10=expert)	Self-rated JS proficiency (10=expert)
avg	7.43	3.37	5.63	4.94	4.69
st. dev	4.15	3.36	2.16	2.59	2.39
min	2	0	2	1	1
max	15	12	10	10	9
mode	N/A	N/A	3	4	4

Table 2.2 Online Study.

	Years of Programming experience	Years of Web Dev experience	Self-rated HTML proficiency (10=expert)	Self-rated CSS proficiency (10=expert)	Self-rated JS proficiency (10=expert)
avg	2.4	0.95	4.9	4.26	3.8
st. dev	2.25	0.85	2.23	2.02	1.81
min	0	0	1	1	1
max	8	3	8	8	7
mode	N/A	N/A	5	5	2

Our experimental group used Jimbo in teams of two. We had 8 participants randomly assigned to 4 teams. Members of a team were seated next to each other, and we asked

them to use their own laptops rather than share a screen. An experimenter was assigned to play instructor and help students remotely with problems using Jimbo.

The task for both groups was to use the d3.js library to create a bar chart to represent a set of data. While all students in the experimental group finished the task in the given time (40 minutes), none in the control group finished. Thus, for our analysis we focused on our qualitative data. This includes observations and interviews to understand the reasons for the experimental group's success and the control group's failure.

Results

In general, all participants agreed that it was more effective for them to learn new topics by working on practical examples compared to the more passive lecture mode:

"[…] back in college […] they would give us the code and we'd compile it and then they'd explain it to us […] while I thought I knew everything, when I wanted to do the homework, I had no idea what the code was about […] but this was my own code, I understand it better"

"I used to think I understood the concept from slides […] when it comes down to using them in code, I have trouble converting it to code […]"

They also liked learning with pair programming, it really helped them to understand concepts and details better. One participant explained that, *"[..] when we have trouble with our homework we go to each other for help anyways, pretty much a collaborative thing […] it helps a lot to have someone work with you on the same project […] I get less stressed".*

The overall experience of the experimental group was mainly positive. The most liked feature was the live preview, which helped them finish their task quicker. One of the participants compared Jimbo to jsfiddle.com and said, *"I use that website a lot but there you have to click run over and over to see the changes in action, less interruptions is better".*

Next, we analyzed the interviews and our observations to better understand the issues in the classroom setting that make it difficult to implement an interactive teaching/learning experience for students. Then we show how the design goals and principles we followed in Jimbo would address those issues.

Web-based Zero-setup Environment

All the teams in the experimental group finished the task, while participants in our control group spent a large amount of time setting up or fiddling with their pre-installed coding environment. This despite being allowed to use their own tools, presumably installed and familiar to them. This shows that a zero-setup tool is essential. Only 3 participants from the control group finished part of the programming task.

Synchronous Communication & Collaboration

Participants indicated that the long wait for help is a major problem in CS classes with programming tasks.

"I usually send them [instructor and TAs] my code in an email to get help […] it's a long process"

"Office hours are very busy; I'd rather solve it myself or get help from my friends […]"

We observed that in our control session the instructor was consistently walking around the room to help students with, and while helping one student, some others were waiting to get help from the instructor, and others again were getting help from other students.

Students in the experimental session were pleased that they could talk directly with the instructor, ask questions, and that the instructor could join their code environment in real-time to help them with their issues. One participant said,

"I asked a question in the chat box and I saw [the instructor] joining my code and helping me, that was awesome!"

Given that they were all collocated, participants in the experimental group mostly communicated verbally, except when they wanted to share a link or code snippet, or they needed to interact with the instructor. They all agreed that voice chat would be a must for them if they are not co-located.

Integrated Live Preview & User Awareness

All participants in the experimental group liked the live preview feature and found that it reduced development time. They also mentioned that whenever they noticed changes in the results they were able to see what line of code was being changed by their teammate. This helped them to coordinate their tasks better and prevent any potential code conflict.

"I could easily see the changes in the preview window and quickly look at the code to see what line is being changed"

"It was not distracting at all, it was very helpful […] I could see what part of the code he is working on […]"

On the other hand, in the control group, we found that most of the students modified the instructor's code while switching back and forth between editor and browser to check the effects of their edits.

2.7.2 Online Setting Study

After our initial validation study, we picked an online course on web development as a second case study. The course requires students to design and develop web sites, and is part of an online post-bachelor course, meaning students have a bachelor's degree in a subject other than Computer Science. The course is offered as an early part of an intensive 1-year program terminating in a second degree in Computer Science. Students are predominantly U.S. based, but are otherwise geographically diverse.

The class follows a fixed term pattern (10 weeks), with regular evaluations. Instruction is a mix of online PowerPoint and other written materials, and video clips prepared by

instructors doing live coding. Compared to most online classes, this course follows a more structured pace, and students interact mostly with teaching assistants through online chat and email when needed. Students pay tuition, and are therefore perhaps more motivated than the average MOOC student, who often ends up dropping out before the end of the course [38].

Our subjects had a wide variety of academic backgrounds, but most had limited experience with programming. In our study, only 20% of participant had any Computer Science background (other university-level coursework or work-related experience), and none had ever used a real-time collaborative tool. Surprisingly, 63% had experience with pair programming.

Setup and Participants

The experimental design was similar to our on-campus study. We recruited 16 students (5 female) and randomly assigned half to an experimental group that would use Jimbo, and a group that continued to work with their preferred tools.

We asked participants in the experimental group to participate in a Google Hangout session where we instructed them in how to use Jimbo. An instructor taught a new concept in web development using live coding. We then randomly assigned participants in the experimental group into teams of two, who then used RPP to complete an assignment using Jimbo. All communication between team members in the experimental group happened inside the tool in order to allow us to track interactions. Our control group with 8 subjects worked on the same task individually using their favorite pre-installed tools.

The task assigned for both groups was to use the NYTimes RESTful api to create a web application that allows users to search for articles based on a keyword and time period. We collected code artifacts from both the experimental and control groups. After the experiment concluded we interviewed all participants to learn more about their experiences, perceptions, opinions, and attitudes.

Results

Two graders scored students' code using a common grading rubric. According to this rubric, 80% of the grade was for following task specifications and delivering the correct program, the rest focused on code readability and organization. We used the interclass correlation method (model: two-way, type: absolute agreement) to check that the graders were consistent (Control group: icc = 0.873, experimental group: icc = 0.965). The final grade for each student was the average of the grades given by the two graders. Median grades in control and experimental groups were 81.75 and 92.5; the distributions in the two groups differed significantly (Mann–Whitney U = 11, n1 = n2 = 8, $P < 0.05$ two-tailed).

We did the same analysis on qualitative data from our interviews and observations for this study as well. In the rest of this section we discuss that how our design goals addressed the issues students currently facing in online programming classes.

Web-based Zero-setup Environment

Several participants remarked that Jimbo was very similar to the tools they usually use, such as Eclipse, Visual Studio and NotePad++, but they liked that they could start coding without any initial configuration or setup work. Others disagreed and mentioned that those configurations are "one time" and were for them not a big deal; *"[…] once you set it up […] at the beginning of the term you are good to go for the rest"*

Synchronous Communication & Collaboration

When asked about current barriers in online classes, subjects unanimously pointed to a lack of good support from instructors when doing programming tasks. One participant explained: *"We get the sample codes from [the] course web site […] we don't know how to run them […] it's due date for the homework before I can get help from TAs or instructor"*. Others strongly agreed with this sentiment. Subjects liked the "connectedness" of learning with Jimbo; *"I pinged the instructor in Jimbo and boom, he was there in my code helping me, awesome!"*

Subjects in the experimental group all used audio chat for communication and shared web resources or code snippets using the text chat. All the teams in the experimental group spend some time at the beginning on coordination and making sure they understand the task. Most participants mentioned that they preferred audio chat to text chat, at least for these tasks.

Subjects reported that coordinating tasks and sharing code is the most difficult part of team projects in their online classes. While some students said that they used version control tools such as git, svn, etc., others reported having difficulties using such technical tools and preferred using file-sharing tools such as Dropbox and Google drive.

The most liked feature of Jimbo was code editing in real-time. Interestingly we observed different patterns of pair programming. Although all students contributed to the code while coordinating their tasks with each other, one team followed a strict model of pair programming where one student was writing the code and the other one was reviewing and pointing out the issues.

Subjects in the experimental group found that situating the chat window next to the live preview streamlined the collaboration process and made development quicker:

"I was surprised how we could finish a homework that I personally spend an entire weekend [on ...] in half an hour"

"[...] if I can find a fixed time with my teammate we can use Jimbo to finish the homework much quicker"

One participant suggested adding a feature to switch between asynchronous and synchronous editing mode.

Integrated Live Preview & User Awareness

While most of the participants like the idea of live preview, some found it distracting and would rather run the code manually; *"I was fixing something in JavaScript and the view disappeared! [...] pretty sure it wasn't me [...] but took a while to figure out it was*

her changing the HTML code". Some recommended adding a pause button for the live preview.

Overall the feedback from students using Jimbo was overwhelmingly positive, though some tweaks and improvements were suggested. We were pleased to hear that all students except one wanted access to the tool for the rest of the term. More importantly, the design requirements identified in section 3 were validated, in that all of them were seen as valuable and contributing to improving the learning experience, especially for the online students.

2.8 Discussion

The goal of our research was to validate a set of design requirements aimed at better supporting students in online settings. Specifically, we wanted to see if we could support remote pair programming and a tighter code-artifact feedback loop. Though most of these concepts have been explored individually in tools or programming environments, such as Syde [21], Collabode [18], CoRED [31], Brackets [5], and Plantír [48], our Jimbo system combined all the necessary requirements, including: live coding, synchronous collaboration, communication and user awareness. We hypothesized that this would allow learners to work more closely together and write code better and faster. The key question for us was whether these features could be combined in a way that would be useful to learners, and whether students in real programming courses would find these features as useful as we expected.

The results from our evaluation show that Jimbo, as an instantiation of these design principles, can be an effective and useful tool in teaching new programming concepts to students using the remote pair programming method. While students were generally happy with the tool, to the point of requesting that the tool should be available after the end of the experiment, we did get some feature and change request from participants.

Although our tool supports traditional pair programming through a shared view of code, the instructors in the classes did not force students to observe traditional pair programming roles. As a result, most students in our studies decided to adopt a looser

collaboration style. Participants simultaneously contributed to the project by writing code, while coordinating their tasks through the provided communication channels such as text and audio chat. Most of our participants stated that they wanted to get their hands on the code when learning a new programming technique, perhaps ignorant of the research showing the positive effects of pair programming. This could explain why they preferred to adapt a looser version of pair programming, where pairs can be both driver and navigator. That said, nothing would have prevented an instructor using our prototype from following a traditional pair programming model of work, regardless of the geographic distribution or size of the class (provided there is time overlap and the class can be broken down in pairs).

We supported several different channels of communication in Jimbo, which enabled collaborators a great deal of flexibility on how they worked together. We did not however determine whether these channels were necessary, or which would be more efficient. Our subjects showed a distinct preference for audio, with text-based chat being a backup for exchanging code and links. Audio communication is of course more bandwidth intensive, and may not always be a good option, especially if audio quality cannot be maintained.

In Jimbo we tried to reduce the efforts of task coordination by notifying users about changes in the project through a powerful push notification system. This system follows the continuous coordination approach introduced by van der Hoek et al. [54] in which users benefit from both formal and informal coordination. These notifications manifested as visible marks in the code file, which would fade over time.

Our subjects embraced the synchronous collaboration and automatic conflict resolution features, explaining that they find current source control systems confusing and hard to use when there is a conflict and the manual merging is required. However, we learned that having a private edit mode is essential to allow students to sync their codes only whenever they feel confident.

The live preview feature provided an instant connection between code and the output for students and allowed them to complete their work more quickly. However, some students explained that the continuous live preview could get distracting. As team sizes grow, this issue could potentially be exasperated, as more edits would mean more visual jumps in the live preview. A simple fix to address this issue could be a lazy update algorithm, which would set update intervals to limit the frequency of interruptions, or a freeze function that users could toggle.

Overall, we found that these features, in combination, but also likely in isolation, are likely to improve the learning experience of online CS students. We believe that the design goals and the main features that we recognized and integrated into our prototype are essential for any tool that wants to support remote pair programming. We also observed the benefits of live coding next to pair programming and our data shows these features play well with each other to enable students master the key programming concepts in their classes.

2.9 Conclusion

In this paper, we explored some technologies and requirements for improving learning in online settings. This includes synchronous communication and awareness features, integrated change tracking and management, a zero-configuration environment, and live preview. We implemented these features into a tool called Jimbo, a web-based collaborative IDE for HTML5 application development.

Through the use of Jimbo in to experimental class settings; one collocated and one online, we confirmed that students found the features we advocated for helpful, engaging, and that it helped them not just engage in pair programming, but also more easily interact with their instructors. As a result, students were able to complete their assignments more successfully and faster, focusing more of their time and effort on the key concepts being taught instead of on syntax and trivial software problems.

2.10 Acknowledgement

We thank our reviewers, participants, and members of HCI research group at Oregon

State University, family and friends for their help and support.

2.11 References

1 Al-Ani, B. Trainer, E. Ripley, R. Sarma, A. Hoek, A. and Redmiles, D. Continuous coordination within the context of cooperative and human aspects of soft- ware engineering. In *CHASE*, pages 1–4, Leipzig, Germany, May 2008.

2 Baheti, P., Gehringer, E. F., and Stotts, P. D. Exploring the efficacy of distributed pair programming. In Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Program- ming and Agile Methods - XP/Agile Universe 2002 (London, UK, UK, 2002), Springer-Verlag, pp. 208–220.

3 Berliner, B. CVS ii: Parallelizing software development. In USENIX Winter 1990 Technical Conference, pages 341–352, 1990.

4 Biehl, J.T. Czerwinski, M. Smith, G. and Robertson, G.G. FASTDash: A visual dashboard for fostering awareness in software teams. In CHI, pages 1313–1322, SanJose, CA, USA, Apr. 2007.

5 Brun, Y. Holmes, R. Ernst, M. and Notkin, D. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (ESEC/FSE '11). Pp. 168-178.

6 Carmel, E. Global Software Teams: Collaborating Across Borders and Time Zones. Prentice-Hall: Englewood Cliffs NJ, 1st edition edition, 1999.

7 Carzaniga, A. Rosenblum, D.S. and Wolf, A.L. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 2001.

8 Cheng, L. Hupfer, S. Ross, S. and Patterson, J. Jazzing up eclipse with collaborative tools. In 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications / Eclipse Technology Exchange Workshop, pages 102–103, Anaheim, CA, 2003.

9 Cliburn, D., Experiences with pair programming at a small college, The Journal of Computing in Small Colleges, 19, (10), 20-29, 2003.

10 DeClue, T., Pair programming and pair trading: effects on learning and motivation in a CS2 course, The Journal of Computing in Small Colleges, 18 (5), 49-56, 2003.

11 Dewan, P. and Riedl, J. Toward computer-supported concurrent software engineering. IEEE Computer, 26(1): 17–27, 1993.

12 Dourish, P. and Bellotti, V. Awareness and coordination in shared workspaces. In ACM Conference on Computer-Supported Cooperative Work, pages 107–114, Monterey, California, USA, 1992.

13 Ducheneaut, N. and Bellotti, V. E-mail as habitat: an exploration of embedded personal information management. Interactions, Volume 8(Issue 5): 30 – 38, 2001.

14 Ebert, C. and De Neve, P. Surviving global software development. IEEE Software, 18(2): 62–69, 2001.

15 Ellis, C.A. Gibbs, S.J. and Rein, G.L. Design and use of a group editor. In Engineering for Human Computer Interaction, pages 13–25, Amsterdam, 1990.

16 Fitzpatrick, G. Kaplan, S. Mansfield, T. Arnold, D. and Segall, B. Supporting public availability and accessibility with elvin: Experiences and reflections. Computer Supported Cooperative Work, 2002.

17 Gaspar, A. Langevin, S. Active learning in introductory programming courses through student-led "live coding" and test-driven pair programming, EISTA 2007, Education and Information Systems, Technologies and Applications, July 12-15, Orlando, FL.

18 Goldman, M. 2011. Role-based interfaces for collaborative software development. In *Proceedings of the 24th annual ACM symposium adjunct on User interface software and technology* (UIST '11 Adjunct). ACM, New York, NY, USA, 23-26.

19 Grinter, R. Using a configuration management tool to coordinate software development. In *CoOCS*, pages 168–177, Milpitas, CA, USA, Aug. 1995.

20 Hancock, C. M. Real-time programming and the big ideas of computational literacy. PhD thesis, Massachusetts Institute of Technology, 2003.

21 Hattori, L. and Lanza, M. Syde: A tool for collaborative software development. In *ICSE Tool Demo*, pages 235– 238, Cape Town, South Africa, May 2010.

22 Herbsleb, J. and Grinter, R.E. Splitting the organization and integrating the code: Conway's law revisited. In Proceedings of the 21st international conference on Software engineering, pages 85–95, 1999.

23 Herbsleb, J. and Moitra, D. Global software development. IEEE Software, 18(2): 16–20, 2001.

24 Herbsleb, J. Atkins, D.L. Boyer, B.G., Handel, M. and Finholt, T. A. Introducing instant messaging and chat in the workplace. In Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 171–178, Minneapolis, Minnesota, USA, 2002.

25 Herbsleb, J.D. Mockus, A. Finholt, T.A. and Grinter, R.E. Distance, dependencies, and delay in a global collaboration. In Proceedings of the 2000 ACM conference on Computer supported cooperative work, pages 319–328, Philadelphia, PA, 2000.

26 Horn, E. M., Collier, W. G., Oxford, J. A., Bond Jr, C. F., & Dansereau, D. F. Individual differences in dyadic cooperative learning. In Journal of Educational Psychology, 90(1), 153. (1998).

27 Kantor, M. and Redmiles, D. Creating an infrastructure for ubiquitous awareness. In 8th IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001), pages 431–438, Tokyo, Japan, 2001.

28 Karolak, D.W. Global software development: managing virtual teams and environments. IEEE Computer Society, 1999.

29 Knister, M.J. and Prakash, A. Distedit: a distributed toolkit for supporting multiple group editors. In Proceedings of the 1990 ACM conference on Computer-supported cooperative work, Los Angeles, CA, 1990.

30 Krahn, R. Ingalls, D. Hirschfeld, R. Lincke, J. and Palacz, K. 2009. Lively Wiki a development environment for creating and sharing active web content. In *Proceedings of the 5th International Symposium on Wiki and Open Collaboration* (WikiSym '09). ACM, New York, NY, USA.

31 Lautamäki, J. Nieminen, A. Koskinen, J. Aho, T. Mikkonen, T. and Englundm, M. 2012. CoRED: browser-based Collaborative Real-time Editor for Java web applications. In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work (CSCW '12). ACM, New York, NY, USA, 1307-1316.

32 Lvstrand, L. Being selectively aware with the khronika system. In Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW (91), pages 265–278, Amsterdam, NL, 1991.

33 Mark, G. Extreme collaboration. Communications of the ACM, 45(6): 89–93, 2002.

34 McDirmid, S. Living it up with a live programming language. In Proc. of OOPSLA Onward pages 623–638, October 2007.

35 McDirmid, S. (2013, October). Usable live programming. In Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software (pp. 53-62).

36 McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002, February). The effects of pair programming on performance in an introductory programming course. In ACM SIGCSE Bulletin (Vol. 34, No. 1, pp. 38-42).

37 McGuffin, L.J. and Olson, G. Shredit: A shared electronic workspace. Technical Report 45, Cognitive Science and Machine Intelligence Laboratory, Tech reports: 45, University of Michigan, Ann Arbor, 1992.

38 MOOCs on the Move: How Coursera Is Disrupting the Traditional Classroom (text and video). Knowledge @ Wharton. University of Pennsylvania. 7 November 2012. Retrieved 23 April 2013.

39 Murphy L., Blaha, K., VanDeGrift, T., Wolfman, S., Zander, C., Active and cooperative learning techniques for the computer science classroom, The Journal of Computing in Small Colleges, 18, (2), 92-94, 2002.

40 Nosek, J. T. ìThe case for collaborative programmingî, Communications of the ACM 41:3, March 1998, p. 105-108.

41 O'Donnell, A. M. and Dansereu, D. F. "Scripted Cooperation in Student Dyads: A Method for Analyzing and Enhancing Academic Learning and Performance," in R. Hartz-Lazarowitz and N. Miller (Eds.) Interactions in Cooperative Groups: The Theoretical Anatomy of Group Learning, pp. 120-141, Cambridge University Press, 1992.

42 OMG. CORBACos: Notification Service Specification v1.0.1. 2002.

43 Perry, D. Siy, H. and Votta, L. Parallel changes in large-scale software development: an observational case study. *ACM TOSEM*, 10:308–337, July 2001.

44 Resig, J. Redefining the introduction to computer science. http://ejohn.org/blog/introducing-khan-cs/, 2012.

45 Rugarcia, A., Felder, R. M., Woods, D. R., & Stice, J. E. (2000). The future of engineering education. I. A vision for a new century. Chemical Engineering Education, 34(1), 16–25.

46 Sarma, A. A survey of collaborative tools in software devel- opment. Technical Report UCI-ISR-05-3, University of Cali- fornia, Irvine, Institute for Software Research, 2005.

47 Sarma, A. Bortis, G. and van der Hoek, A. Towards supporting awareness of indirect conflicts across software con- figuration management workspaces. In *ASE*, pages 94–103, Atlanta, GA, USA, Nov. 2007.

48 Sarma, A. Noroozi, Z. and van der Hoek, A. Palantír: raising awareness among configuration management workspaces. In *ICSE*, pages 444–454, May 2003.

49 Schneider, K.A. Gutwin, C. Penner, R. and Paquette, D. Mining a Software Developer's Local Interaction History. MSR 2004, Edinburgh, 2004.

50 Slavin, R. E. “Research on Cooperative Learning and Achievement: When We Know, What We Need to Know,” Contemporary Educational Psychology, 21, pages 43-69, 1996.

51 Stefik, M. Bobrow, D.G. Foster, G. Lanning, S. and Tatar, D. 1987. WYSIWIS revised: early experiences with multiuser interfaces. *ACM Trans. Inf. Syst.* 5, 2 (April 1987), 147-167.

52 Sun, C. and Ellis, C. Operational transformation in real-time group editors. In *Proc. Computer Supported Cooperative Work*, pages 59–68, 1998.

53 SunMicrosystems. Java Message Service API. 2003.

54 Van der Hoek, A. and et al. Continuous coordination: A new paradigm for collaborative software engineering tools. In Proceedings of Workshop on WoDISEE, Scotland, 2004.

55 Vessey, I. and Sravanapudi, A. P. Case tools as collaborative support technologies. Communications of the ACM, vol. 38:83–95, 1995.

56 Waldrop, M. Online learning: Campus 2.0 http://www.nature.com/news/online-learning-campus-2-0-1.12590

57 Warren, J. Rixner S. Greiner, J. and Wong, S. 2014. Facilitating human interaction in an online programming course. In *Proceedings of the 45th ACM technical symposium on Computer science education* (SIGCSE '14). ACM, New York, NY, USA, 665-670.

58 Werner, L. L. Hanks, B. and McDowell, C. 2004. Pair programming helps female computer science students. J. Educ. Resour. Comput. 4, 1, Article 4 (March 2004).

59 Williams, L. A., ìThe Collaborative Software Process PhD Dissertationî, Department of Computer Science, University of Utah. Salt Lake City, 2000.

60 Williams, L. and Kessler, R. R. “Experimenting with Industry’s ‘Pair-Programming’ Model in the Computer Science Classroom,” Journal on SW Engineering Education, Dec. 2000.

61 Williams, L. A. and Kessler, R. R. (2000, March). The effects of" pair-pressure" and" pair-learning" on software engineering education. In Software Engineering Education Training, 2000. Proceedings. 13th Conference on (pp. 59-65).

62 Wouters, P. van Nimwegen, C. van Oostendorp, H. and van der Spek, E. D. A meta-analysis of the cognitive and motivational effects of serious games. Journal of Educational Psychology, 105(2):249, 2013.

3 Improving Developer-Designer Collaboration Using Collaborative and Live Programming

Soroush Ghorashi, Carlos Jensen

School of EECS
Oregon State University
Corvallis, Oregon, 97331, USA
{ghorashi, cjensen}@eecs.oregonstate.edu

Proceedings of the 2018 ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '18)
(to be submitted)

3.1 Abstract

Software development involves teamwork and a lot of communication. Traditional software engineering processes are often implemented in an inflexible and change-resistant manner. While these methods worked well in the past, they may not always be the most efficient today. Methodologies such as collaborative programming have emerged, where developers collaborate with other developers, designers, and end-users to create software. Successful product development requires effective collaboration between all three. This leads to an increased need for feedback and checking that the project is progressing in the right direction. To address the needs of developers and designers, we introduce Jimbo, a web-based HTML5 development environment that integrates collaborative editing, live programming and communication features to close the communication gap between designers and developers. Jimbo allows designers to join the development process as first-class citizens – not dependent on the coders to compile and share output – or being forced to become coders.

3.2 Introduction

The development of software systems is a highly collaborative process, bringing together programmers, designers, managers, etc. to solve a problem by producing quality code. Vessey and Sravanapudi [37] showed that about 70% of a typical software engineer's time is spent on collaborative activities. When working with non-developers, collaboration is even more complicated, as cultural barriers must be negotiated. This includes working with designers, end-users or other stakeholders, even developers from other cultures. In addition to the usual face-to-face interactions, software engineers use a variety of tools to streamline this process and decrease the effort involved. Most of these tools are not directly integrated into their development environments (Skype, IRC, email, etc.), which can lead to disconnection between code and design, or simply wasted effort through frequent context switching.

Currently distributed collaborative software development revolves around working in parallel on separate copies of the code, and integrating the resulting efforts using a source code version control system such as Subversion [35] or Git [13]. Though this is an effective strategy, there is a lack of real collaboration, as developers largely work independently, only coordinating when synching code.

Better designer-developer collaboration usually leads to a better user experience in the product [4]. One of the common challenges faced by mobile and web developers is the need to bridge the gap between designers and developers. This applies to many UI or interactive projects, where a tight integration of code and design are integral to the products' success. Current integrated development environments (IDE's) do not offer an effective mechanism for direct collaboration between designers and developers involved in a common software project. Direct collaboration and effective communication results in faster development, better user experience and eventually the success of the software project.

Current software development workflow models require developers to write code "blindly", only seeing whether their changes have the desired outcome after they compile and execute their programs. If they want to further change the code or tweak their designs, they have to go back to the code, edit, compile and run the program again. The problem is that coding is based on a mental image held by developers, which needs to be translated to code.

Developers often spending significant amounts of time switching between code and the build process. This disjointed process is even more painful when working with multiple developers, or with designers on UI-heavy applications.

The rest of this paper is organized as follows: We start with a review of related work. We then describe our tool and explain how it addresses various issues that developers and designers face in collaborative environments. We then present the results of a user study that we have conducted to evaluate our tool in professional development setting. We conclude with a discussion of challenges to developing collaborative tools.

3.3 Related Work

One of the key factors impacting how successful a team can be, is the flow of communication between team members [8, 22]. In a software team, communication between designers and developers is critical. A reliable communication channel leads to a successful project while lack of one can lead to more iteration, a bad user experience and eventually failure. The most common type of communication is the face-to-face meeting. However, as teams get larger and/or geographically distributed, collaboration in software development gets more challenging [11, 18]. Researchers have tried to overcome this by mimicking physical meetings in a virtual environment such as video conferencing systems [18, 20, 24], or more lightweight communication and awareness systems such as email, instant messaging (IM) [19], or even the source control system itself [6, 12]. Of these, email is by far the dominant mechanism due to its low learning curve and flexibility.

In the traditional model of software development, developers get design documents from designers and try to bring these to life. This model breaks down into three discrete activities: Coding in an IDE, compiling/testing the code, and execution to verify that it functions and looks as envisioned in the design documents. As changes are made to the code, developers have to repeat the compile-execute cycle to confirm the effect of code changes as well as designers verifying that their vision is being fulfilled and the development is on the right track. Because developers often need rapid feedback, these cycles tend to come frequently, resulting in inefficiencies. Furthermore, the designer is often out of the loop, providing input at the beginning, and then often depending on the developers to keep them up to speed.

Researchers have recently started investigating the benefits of using live programming in IDE's, an old but increasingly popular technique in software development. As the name suggests, live programming is an approach where programmers can re-execute a program continuously while editing [15], immediately seeing the results of their efforts.

Tanimoto introduced live programming [34] for visual languages in 1992 by defining four different levels of "liveness" which categorizes the immediacy of feedback that is automatically provided during programming. At level 1 the user will not receive any feedback. At level 2 no automatic feedback will be provided to the user, however they can obtain feedback about a specific portion of their code manually. At level 3, the system provides feedback automatically and incrementally upon any changes made to the code. At level 4, the system updates the display to show the results of continuous data processing. Level 4 responds to the user's edits in a way similar to level 3, in addition to responding to system events such as mouse events and system clock ticks.

Victor argues that live programming is promising because it reduces the temporal and perceptual gap between program development and code execution, but is not a replacement for traditional debugging techniques [38]. While live programming provides users with a copy of the final output, it does not provide a mapping between the output and code, as done by step-based debugging or tracing with print statements [38]. Researchers are trying to enhance live programming environments by focusing on debugging [25]. Some recent live programming systems include Superglue and Flogo II [16, 26].

A number of tools have been developed to provide real-time awareness of code changes to facilitate coordination and emerging conflicts. FASTDash [2] and ProjectWatcher [32] provide various visualizations of data directly gathered from developers' workspace. Palantír [1, 29, 30] shows who is modifying which part of the code and alerts developers about emerging conflicts. Syde [17] follows the same approach to provide change information for interested developers but reduces false positives through abstract syntax tree (AST) analysis. Crystal [5] proactively watches developers' code and precisely identifies and reports conflicts.

Some efforts have been made to address some of these issues, most notably Plantír [30], Syde [17], Collabode [14] and Brackets [3]. However, most of these tools have been developed with a focus on developers without considering designers' needs. Thus, we

feel that there is a need to explore this space in order to improve developer-designer collaboration process. We therefore introduce our own tool, Jimbo.

3.4 Developer-Designer Collaboration

Behind every successful software project lies a good UI design that eventually leads to a good user experience. Although having a good design is necessary for the success of a software product, it is not enough, it must be successfully implemented. In order to accomplish this, there needs to be effective communication of the design between designers and developers. This collaboration sounds easy in theory, but is a major challenge that software teams face today: how to foster effective developer-designer collaboration?

To better understand what are currently seen as effective developer-designer collaboration methods, we did a survey of tools and techniques described in the literature. We started our search with the following conferences: CSCW, CHI, ICSE, and keywords; live coding, design tools, developer designer collaboration, pair programming, incremental model, and prototyping. After filtering false positives, we identified 30 papers. From there we refined our list by looking up cited works. From this literature review we identify a set of key practices and techniques that should lead to more effective collaboration between developers and designers. In the rest of this section we review these practices.

3.4.1 Support both synchronous and asynchronous coding for collaboration between multiple developers and designers

Software teams usually have more than one developer and designer collaborating. Instead of isolating designers from the development, we suggest including them in the process and work with the developer while these work with the code. Many designers nowadays have basic development skills, most notably HTML, CSS and JavaScript for prototyping. They can collaborate directly through code instead of sitting outside of the process, giving feedback when asked or showed output by developers. This way,

designers get to guard their designs themselves and developers are forced into the habit of getting more immediate and meaningful feedback.

Having synchronous coding integrated into IDE tools enables designers to make modifications directly, or work with the developers and direct their development efforts without going through a long set of steps to deliver their ideas to the engineering team effectively. This can facilitate a more design oriented process in which designers create the main skeleton for the software product and then the developers fill in the gaps to bring the design to the life.

However, asynchronous coding is also important the facilitate the trial and error efforts often associated with prototyping before settling on a final design.

3.4.2 Communication methods

The integration of communication features into the IDE could help developers and designers discuss and resolve the issues they may face without losing focus on the code. We recommend the following different but equally effective means of communication between developers and designers:

Audio chat

An audio chat system provides virtual presence and makes it easier to coordinate and collaborate. Audio discussions are quick in nature and volatile. This type of communication allows for quicker resolution in case of minor design misunderstanding between developers and designers that if stay unresolved, it sometimes will lead to major defects in the product and delay in the process and in worst case failure of the project.

Text chat

A text-based chat or instant messaging is a complimentary system for audio chat and is a must, as coders share code snippets and links to resources.

Inline discussions

It is important to recognize that any communication, formal or informal, requires common knowledge in order to adequately interpret messages communicated. Inline discussions are text-based semi-synchronous way of communication, in which users can associate a short discussion with an artifact such as a portion of code or a design document. Unlike transitory audio discussions, inline discussions are tied to people and artifacts and tend to be permanent, having their own context that will not be lost over time.

3.4.3 Integrated live preview to support designer involvement in the development process

Live preview is a live programming technique that shows the code output immediately upon a change to the code, and it best fits UI-heavy application development such as websites. In an environment where designers and developers are collaborating with each other, live preview is a powerful weapon for designers.

Using live preview, designers can instantly see what changes developers are making and provide feedback and directions quickly, or warn the coding team if they are deviating from the design vision. It also streamlines the flow of communication within the team, and make it easier for developers to ask designers for input or help when changes have to be made.

3.4.4 Support awareness of the activities of other in order to facilitate collaboration

A very important aspect of collaboration is awareness. Dourish and Belloti define awareness as “an understanding of the activities of others that provides a context for your own activity” [9]. Awareness is required to coordinate teams, but can be distracting if it interrupts or requires too much attention from members. It is not a very easy issue to address in coding, as we juggle the need for asynchronous editing for some developers, and the need for real-time preview of the resulting code for others.

A good awareness system in a collaborative environment will provide useful answers to the questions that a user may have, for example:

- Who made what change in the code?
- Who should I contact if I have a question about this part of design/code?
- Who is currently available? What are they doing now?

We seek to integrate all these features into a single IDE tool. Our goal is to improve the collaboration within software teams, groups of people with common goals, to help them achieve a better user experience in their product by facilitating the involvement of designers in the development process as the first-class. Our focus is on the development of web applications that requires constant interaction between designers and developers on the daily basis. This means that our focus is on the developer-designer collaboration, communication and coordination not the extended team, which includes management and support, nor the customers and marketing personnel. In the next section, we describe our tool Jimbo, that combines all the key features mentioned above.

3.5 Jimbo Overview

Jimbo is an IDE that enables developers and designers to more easily collaborate around a common project. We have tried to make the user interface easy to learn and memorable, but have also considered external consistency with other popular IDEs. Jimbo is a web-based IDE, which means that users only need a standard web browser and the setup time is zero. Developers with different levels of experience, from novice to expert, may use this tool to create web apps. Figure 3.1 gives an overview of our system architecture. Jimbo follows standard client/server architecture.

Sarma provides a comprehensive classification of collaborative tools for software development [31]; Jimbo could be considered a seamless tool at the top level as it provides many novel features to automate the development workflow and minimize user efforts. In the following sections, we describe these features.

3.5.1 Main View

In Jimbo, users open the IDE in the browser to select and open the project they want to work on. Only a user with enough authority e.g. a manager, can add or remove users to projects. Figure 3.2 shows the general structure of the Jimbo's UI.

On the top left, there is a standard menu with many options available for customizing different parts of the IDE. The left side of the screen is dedicated to a tab view that contains project file structure, chat list, project settings and online deployment options. On the right side, users can turn live preview on and off by clicking on the eye icon. This option, if selected, will open a live preview panel where users can see the rendered version of the current HTML file. A live preview of the entire web app is available through the top menu. Next to that is the notification center where all the notification sent to a user can be access through this option. Once a user opens a project, they can access the project file structure represented as a file tree in the first tab. Anybody in the project can add or remove files or folders to it by right clicking on the desired folder and selecting the right option (add or remove) in that tree. Second tab shows who is currently available in the project and what file they are currently working on if any. By clicking on any name, they can start a text-based chat session with that person or they can select an option to talk to everybody in the project. Jimbo keeps the chat history for those people who are offline and enables them to access them later. Chat windows will pop up on the bottom left corner and stack horizontally to the left in case of more than one active sessions.

The code editor is located in the middle of the screen. Selecting any file will open that file in the code editor and the breadcrumb navigation on the top of the editor, shows where that file is located in the project. At the heart of any popular and successful IDE sits the code editor. Modern code editors include a number of important features designed to scaffold the task of programming. This includes features such as syntactical highlighting, function completion and inline documentation, automatic indentation and even auto-correction and spell checking.

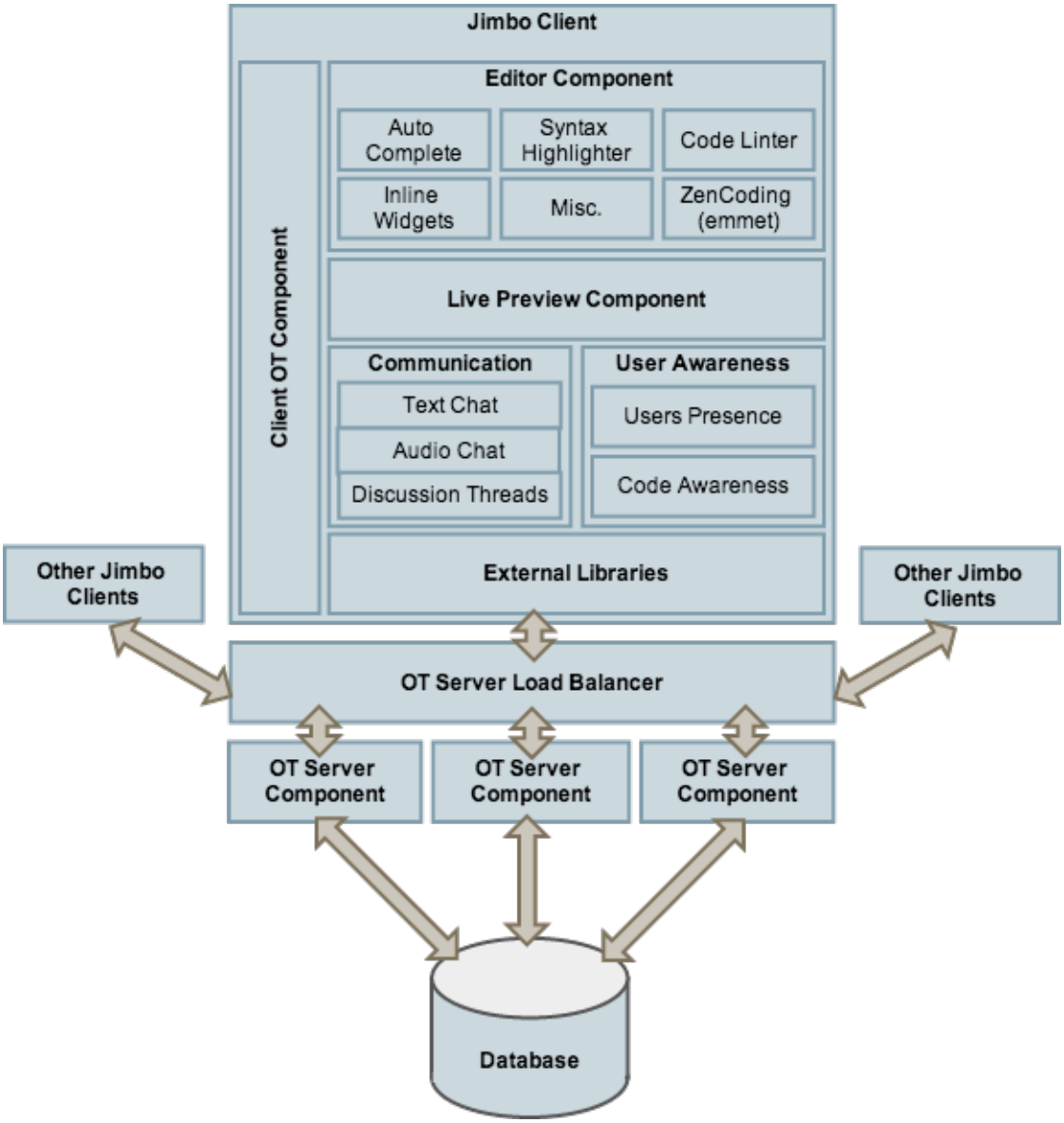

Figure 3.1 Jimbo's system architecture.

All of these features are aimed at helping the users focus on the logic of the program rather than the minutia of syntax. Users can initiate a discussion about any portion of the code by selecting that code and right clicking on it. Lines with an icon on their left gutter indicate these discussions.

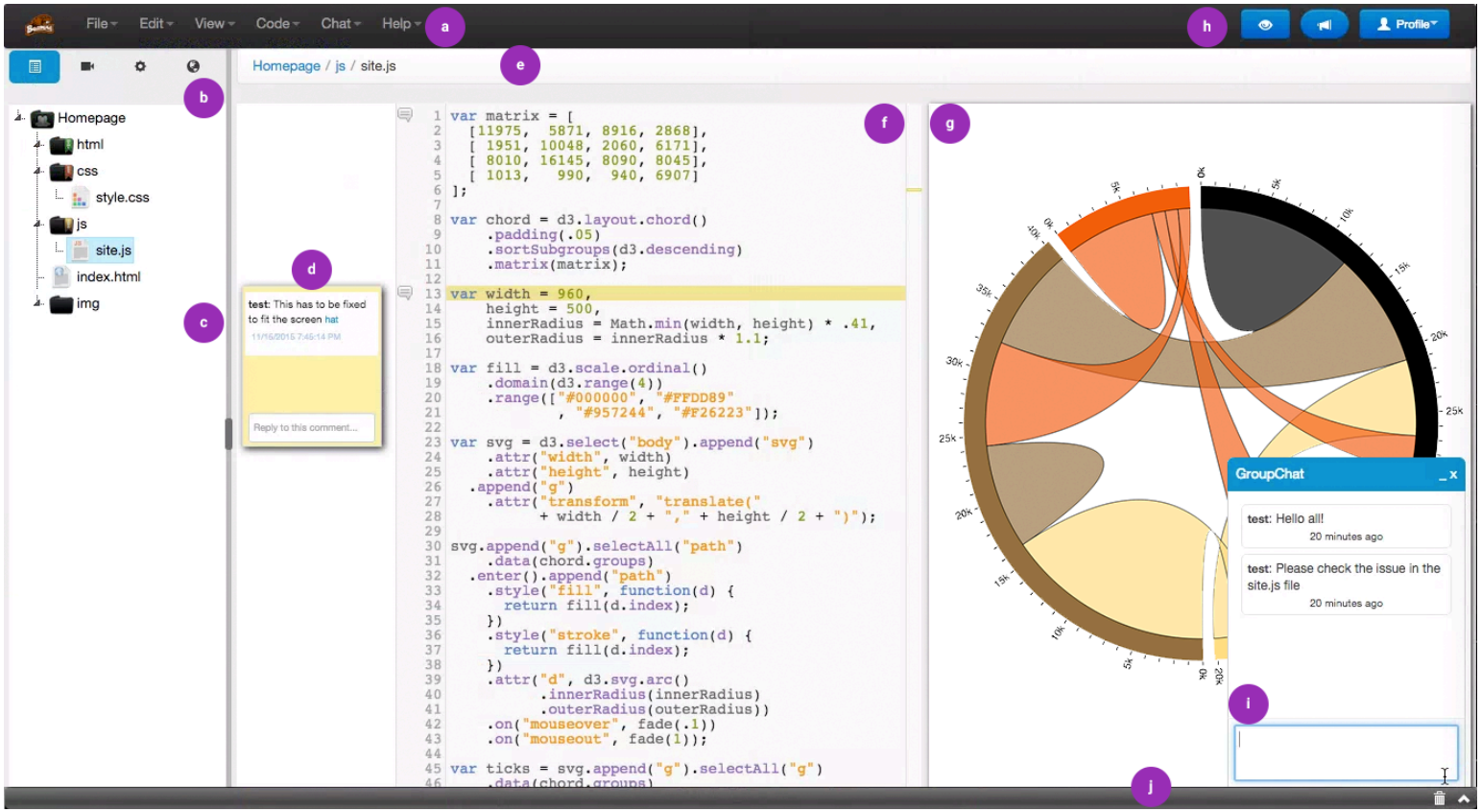

Figure 3.2 (a) top menu. (b) Left sidebar panel that contains the following: file tree, chat list, project settings and online deployment. (c) Current project file tree. (d) Inline discussion with user tagging feature. (e) Breadcrumbs view showing current path. (f) Editor viewport. (g) Live preview panel. (h) Live preview toggle and notification center. (i) Group chat popup. (j) Console.

Finally, on the bottom of the screen, users can access the console. It contains the error and warning messages regarding the syntax issues in the code. Jimbo is equipped with JavaScript, CSS and HTML linters that watch the code and update the console upon any change to show the syntax issues for open files if there are any.

3.5.2 Synchronous and Asynchronous Collaboration

The most important feature of Jimbo is synchronous collaboration. The number of defects in code tends to rise with the amount of parallel work [28] and users sometimes avoid this kind of development to avoid having to resolve conflicts [15]. To minimize problems, Jimbo acts such as a real-time Git tool using an Operational Transformation (OT) algorithm [33].

OT is a technique that provides eventual consistency between multiple users working on the same artifact without retries, errors, or data being overwritten. As the collaborators edit their code files, the client OT component generates mini commits to the code called operations that can be either insert or delete. These operations get transmitted to the server sequentially to be applied to the shared document. In the basic form, the server keeps a state space per connected client, which can be memory-intensive and can make the transformation algorithms complicated.

In Jimbo however, we made the process simpler and more efficient by requiring the server to acknowledge clients' operations before they can send new ones. This means that clients can have only one non-acknowledged operation in flight and the client OT stores other users' operations in the local storage of the browser, only sending the next when the last in-flight operation has been completed. Thus, the server only needs to keep one state space for all connected clients.

The server also keeps different snapshots of each code file. If multiple collaborators try to edit the same code at the same time, one of the edits will be received and applied first and then the server transforms and applies the other edits using the state space. Unlike common revision control systems (e.g. Git), OT algorithm makes sure that the commits and updates happen automatically and the code files are consistent for everyone in the

project no matter in what order the operations are applied to the shared code file. This method enables developers to work offline and then sync their edits with server later; the OT algorithm takes care of the conflicts.

3.5.3 Code Rewind

This feature enables users to see what changes were made since last time they contributed to the project and step through them using incremental snapshots (Figure 3.3). Each snapshot contains all the source code written at any given timestamp.

The slider on the top of the view allows users to step back and forth in time. The overview on the right side of the window provides users with a big picture view of the snapshot they have requested. The content is color coded to show who edited what part of the code.

Developers can use this feature to see how others are contributing to the project, maintain situational awareness, and identify whom they need to communicate with based on code ownership. Another use of this feature is to determine the provenance of the bugs introduced into the code either by other developers or through copy/paste reuse.

3.5.4 Communication

Jimbo explores novel ways of integrating both synchronous and asynchronous communication. Jimbo implements discussion threads, a semi-synchronous communication method. These threads are associated with specific lines of code as inline comments. This allows developers to add and preserve contextual and design information, often generated in discussions with collaborators. Users can also tag who they want to discuss a line of code with as a way of directing questions, and Jimbo lets you know if others have contributed to the discussion thread.

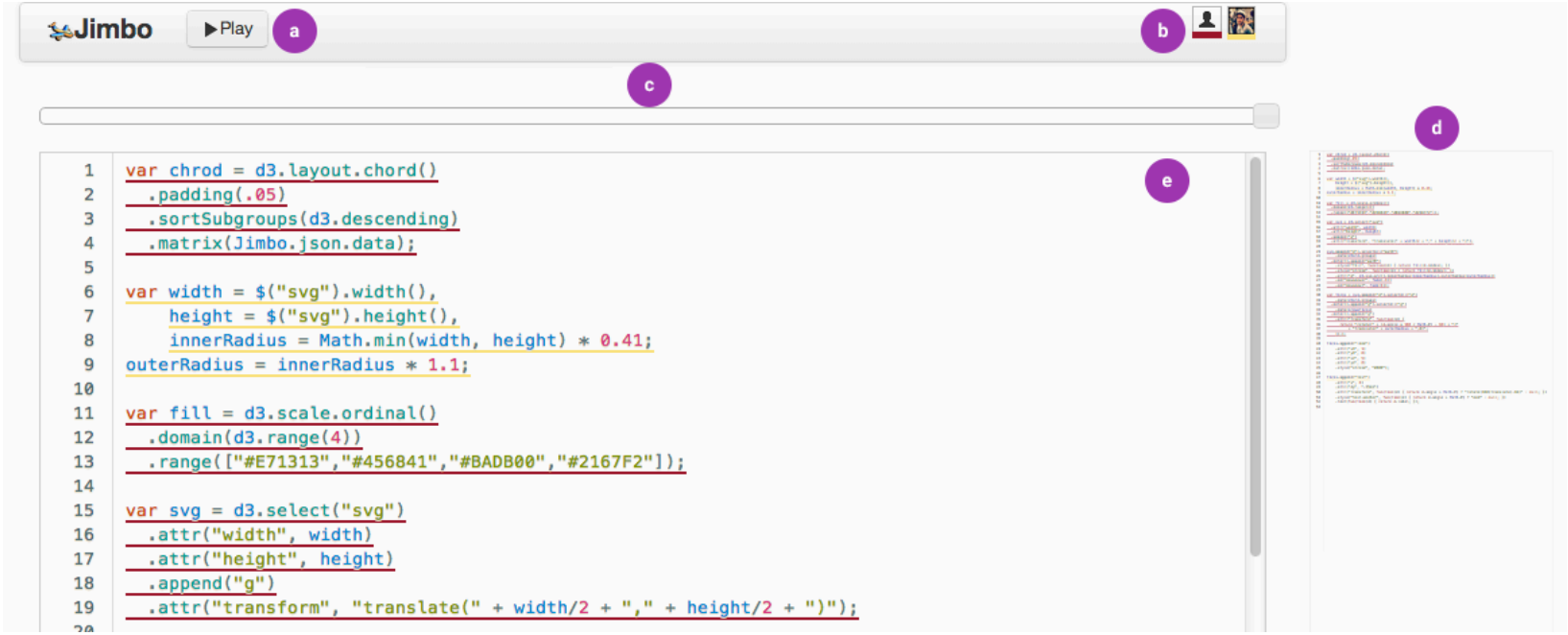

Figure 3.3 Code rewind: (a) Playback button to view the evolution of the code. (b) List of collaborators who contributed to the code; moussing over each will show the percentage of their contribution. (c) Timeline for the life of code; users can go back in time, undoing and redoing changes to the code using this slider. (d) Code overview. (e) Actual code, the colors show who has developed which parts of the code. No colors means that the code has been copy pasted form a resource out of the IDE.

3.5.5 User Awareness

The main purpose of an awareness system is to help coordinate tasks. We follow the "continuous coordination" model introduced by van der Hoek et al. [36]. The primary responsibility of such a system is to notify collaborators of events relevant to them, such as code changes, comments to discussion threads, user presence, etc.

Jimbo has a channel based notification system [27] using push notifications [7, 12]. These are persistent and stored on the server for future retrieval [21, 23]. To prevent cognitive overload, developers can request to only receive notification about a specific portion of the code, a feature we call "code watch". Once someone puts a watch on a portion of code, Jimbo only pushes notifications regarding changes to that section of code. This allows developers to keep track of code they depend on, or code that they have some ownership over.

Users editing the same file can see each other's cursors. This way users are notified about changes in the code that lead to a change in the preview panel.

3.5.6 Live Preview

This feature as we mentioned before will be the ultimate weapon for designers in their closer collaboration with developers. It provides an immediate connection between the code and the output, so the designers can provide feedback for the changes to the code. This also streamlines collaboration between developers, designers, and end-users by providing a better common knowledge in the communications between them and leads to fewer iteration of the code, which means faster coding.

This feature, when enabled, only allows safe and error free code to be run in the live preview. This means that as long as there are syntax errors in the code, Jimbo's built in linter won't allow the code to be run. In this case, the live preview panel will show the last correct code. Collaborators can turn off this feature to mitigate any distraction at any time they want.

Jimbo also has some helper widgets for numerical and color values which allows users to modify values using sliders and pickers instead of typing. This allows specially designers to modify the code written by developers and quickly see and discuss the continuous effects of their modifications in the preview panel.

3.6 Evaluation

In order to evaluate whether Jimbo's design goals will be useful in practice, we conducted a between-group user study in which pairs of web developers and designers worked remotely for 60 minutes to complete a web design and development task.

Table 3.1 Experience in years for web developer participants.

	Years of Programming experience	Years of Web Dev experience	Self-rated HTML proficiency (10=expert)	Self-rated CSS proficiency (10=expert)	Self-rated JS proficiency (10=expert)
avg	6	3.64	6.93	5.14	7.21
st. dev	3.28	1.6	1.77	2.38	1.58
min	2	1	4	1	4
max	14	6	8	8	9
mode	N/A	N/A	8	6	7

Table 3.2 Experience in years for web designer participants.

	Years of Design experience	Years of Web Dev experience	Self-rated HTML5 proficiency (10=expert)	Self-rated CSS proficiency (10=expert)	Self-rated JS proficiency (10=expert)
avg	4.29	3.3	7.14	7.71	3.5
st. dev	1.7	1.27	1.92	1.33	1.45
min	2	2	1	5	2
max	8	6	10	10	9
mode	N/A	N/A	8	7	6

We recruited 28 professional developers and designers (6 females). Out of 28 subjects, half identified themselves as web developers with the average experience of 6 years in general programming and 3.64 years in web development. The other half, identified

themselves as web designers with the average experience of 4.29 years in web design and an average of 3.3 years in web development. The average self-reported age of participants was 29. All but one participant held a bachelor's or higher degree, and all but 4 majored in computer science. All participants indicated that they have prior experience with remote collaboration, mostly using a distributed version control system (e.g. git tools). Participants were asked to rate their proficiency in HTML5, CSS and JavaScript on a scale of 1 to 10 (10 being the most proficient), their responses are listed in Table 3.1 for the developers and Table 3.2 for the designers.

We randomly assigned participants into pairs so that each pair consisted of one web developer and one web designer (14 pairs). Then, we randomly assigned each pair to be in either the control or experimental group.

Pairs in the control group were asked to use their favorite pre-installed tools while the experimental groups used the Jimbo IDE to complete the given programming task. All participants were geographically distributed and had to collaborate and communicate with each other remotely.

3.6.1 Task

Participants were explicitly told that they could use any web resources in order to complete the task and that they could organize their work any way they liked. All participants used Google to conduct searches.

The task for both treatments was to use HTML5, CSS and vanilla JavaScript to create a web-based "To Do App" with the following requirements:

1. App should be viewable on any standard browser on a mobile device.
2. A fixed header on top with app name or logo on it.
3. Users must be able to add new tasks.
4. Each task has 4 fields: title (required), description (not required), creation date (required) and status (new or done, required)
5. Users must be able to view task details.
6. Users must be able to edit their tasks.
7. Users must be able to delete a task.

8. Users must be able to filter their tasks based on the status.
9. Users must be able to do keyword search on their tasks.

Participants received the task description in an online document. We also included the rubric that would be used to grade their code, so they could maximize their grade. To makes sure that everyone understood the requirements, we briefly discussed these with them.

Pairs in the experimental group received a tutorial on Jimbo alongside a warm-up task before they started, lasting approximately 15 to 20 minutes. After that, pairs started working on the programming task, which took 60 minutes.

3.6.2 Data Gathering

To facilitate our data analysis, we gathered multiple data sources from our participants during the study session:

- Screen captures – recordings of participants' screen
- Communications (audio and IM)
- Code files and design documents
- Questionnaires and interviews

3.7 Results

Code Quality

We started our analysis by measuring the task completion and the code quality. Two graders scored each pairs' code using a previously given grading rubric. This rubric, defined a score for each requirement listed in the task based on the complexity of the requirement. We used the interclass correlation method (model: two-way, type: absolute agreement) to check that the graders were consistent (control group: icc = 0.989, experimental group: icc = 0.944). The final grade for each pair was the average of the grades given by the two graders. Median grades for the control and experimental groups were 65 and 82.5 respectively (out of 100). Our analysis showed that the distribution in the two treatments differed significantly (Mann-Whitney U = 1.5, p-value = 0.0041, z-score = -2.875).

Number of Design Iteration

One of our initial hypothesis was that the integration of collaborative coding and live preview would lead to a lower number of design iterations, and eventually faster development. To validate this hypothesis, we analyzed the screen recordings of the pairs working on their task. We used an open coding approach and two independent coders to categorize participant pairs' actions. This way we were able to count the number of design iterations each team went through before the end of the study.

We used the interclass correlation method (model: two-way, type: absolute agreement) to check the consistency between our coders (icc = 0.976). We used the average number of design iterations observed by our coders for our analysis. The median number of design iterations for the control group teams was 4, and 3.5 for the experimental group teams. Our analysis showed that there is no significant difference in the number of design iterations between the two treatments.

Any round of design iteration required the attention of both team members that could have led to a possible interruption in their flow. Our coders recorded these interruption during our screen recording analysis (icc = 0.933). The median number of interruptions were 7.5 and 3 for control and experimental treatments respectively. Our analysis shows that the distribution in the two treatments differed significantly (Mann-Whitney U = 4, p-value = 0.01, z-score = -2.55).

Collaboration Style

We did not coach the participant teams on how to collaborate. On the contrary, we wanted to observe which collaboration style they would adopt given that the control group teams had access to any tool they wished, while the experimental group teams only had access to Jimbo. We used our screen recording analysis to categorize teams' style (icc = 0.912).

All expect one team in our control treatment applied the traditional "individual effort/merge later" approach. In this approach, pairs do an initial duty assignment at the beginning, and merge their individual efforts periodically. They spent 57% of their time

on individual development and 20% on merging their efforts on average. One team however, used the traditional pair programming approach in which they shared their screen and worked on the task in the driver-navigator roles.

In the experimental treatment, all teams used the pair programming approach. However, while one team deployed the traditional version, others deployed the remote pair programming method in which both team members contributed to the code simultaneously. They spent 70% of their time on collaborative tasks and only 12% their time working individually on average. Their individual efforts included online search, reading the task requirements and reviewing the design document.

Communication

All the teams in the experimental group used the audio chat offered by Jimbo to constantly communicate with each other. In fact, on average, they spent 83% of their time talking to each other. They also occasionally used IM to share links to online resources. On the other hand, teams in the control group on average spent only 37% of their time communicating. They used a variety of communication channels including IM, audio and video. IM was used constantly while other methods were only used if the subjects felt it was required. For example, one group used the video channel to facilitate traditional pair programming through screen sharing.

We were pleased to see that the participants in the experimental treatment gave Jimbo an average rating of 4.64 using a 5-point Likert scale (min = 3, max = 5, $\sigma = 0.6$) when asked to rate their overall experience with the tool. While the most liked feature was the collaborative coding (7 votes), live preview came a close second (6 votes). On the other hand, the least liked feature was the code editor itself. 10 subjects in the experimental group mentioned that if they had to use Jimbo for their daily tasks, its code editor needs to match the features offered by popular IDEs such as WebStorm, Sublime, Atom, etc. One developer took it further and said: "*[the] ideal solution would be implemented as a plugin for existing IDEs (in our case Visual Studio, but it could be Eclipse as well). This*

allows developers to continue using an IDE they are familiar with and leverage 3rd party IDE extensions too".

3.8 Discussion

The goal of our research was to determine if the identified design goals could improve how developers and designers collaborate with each other in a designed-centered software development setting. Our hypothesis was that the combination of synchronous coding and live preview, supported by good communications and user awareness would improve collaboration in a way that allowed designers to get more involved in the development process. The main research question was whether professional developers and designers would find these features useful and effective.

Although we did not coach subjects in our experimental treatment on remote pair programming, all but one pair applied this approach to complete their task. As a result, designers were more involved in the development process, where they mostly focused on the CSS and HTML (UI) part of the app. Developers mentioned that having designers contribute directly to the production code, allowed them to be more focused on the logic of the app. One developer said: "*Having my [designer] teammate taking care of the view part of the app made it easier for both of us, we were focused on what we do the best*".

Our subjects were excited about the collaborative coding feature and how it might decrease the number of merge conflicts: "*Whenever someone makes a change, my source is automatically updated in real-time and auto merge is performed. Under some cases I may need to do manual conflict resolution. I'm thinking this would reduce the number of conflicts overall since merges are happening much more frequently.*" However, some developers mentioned that sometimes they would like to contribute to the code asynchronously, meaning that they would like to work individually and commit their changes whenever appropriate: "*For me I would personally prefer an optional manual commit mode, where others do not see my changes until I manually commit […] perhaps this is similar to Jimbo's offline mode*"

Designer participants in the experimental treatment found it much easier to use live preview when communicating the design to the developers: "*instead of discussing it over the initial design doc we came up with, I explained it right there in the preview*". Participants in the control group found the frequent context switches between the output and the code annoying when collaborating in real-time: "*we spent a lot of time jumping between the [HTML] view and the code for our controllers and models*". Our study shows that the live preview can provide a common language for designers and developers to streamline their collaboration when they are discussing the app design.

Jimbo offers two communication channels for users to talk with each other: Audio and IM. Teams in the experimental treatment used audio chat for constant communication and coordination, however all the participants including control groups mentioned that in real-world settings, they use IM for communication and face-to-face meetings or phone calls for project planning and task coordination: "*Team members meet or [use] phone or skype to define a plan [...] In the event one developer needs help or feedback it would typically happen over IM or screen sharing and phone*".

While participants in the control group reported some confusion over the status of the other team member's efforts, experimental group participants mentioned that they were able to monitor each other's status through the user awareness features offered by Jimbo: "*I could see some kind of visual indicator [...] it helped me realize [that] someone else is working on the same file as me and we could collaborate/chat as necessary*". User awareness in Jimbo is done through push notifications and is implemented based on the continuous coordination model described by van der Hoek [36].

The focus of our experiment was on web platform, however with the growth of smartphones and demand for phone apps in the past decade, we can see a use case for Jimbo in that area, where the live preview can switch between different phone simulators, and developers can test their apps on the many devices is in the market before shipping their apps. We believe that Jimbo, or the approach explored in Jimbo,

can be used in other popular programming languages such as Java, python, etc. where live preview can show the content of the data structures and variables at any point in life time of the code running similar to probing concept discussed in [25].

3.9 Conclusion

Jimbo is a collaborative IDE tool for developing web applications. Jimbo streamlines the traditional workflow of developing web-based application by providing a live preview of the output of the code in which users can immediately see what they are doing. This way, designers can be more involved in the development in a more meaningful fashion. Jimbo allows multiple users to work on the same code synchronously and it takes care of potential conflicts on the fly.

Through a user study evaluation, we were able to confirm that a development tool similar to Jimbo, that offers an integration between the collaborative coding and live preview supported by proper communications and user awareness features can close the gaps between developers and designers in a way that they can collaborate closer than ever to produce a higher quality web app in less amount of time.

3.10 Acknowledgement

We wish to thank our colleagues in the HCI group at the School of EECS, Oregon State University for their support and help in preparing the paper. We would like to thank our study participants.

3.11 References

1 Al-Ani, B. Trainer, E. Ripley, R. Sarma, A. Hoek, A. and Redmiles, D. Continuous coordination within the context of cooperative and human aspects of soft- ware engineering. In *CHASE*, pages 1–4, Leipzig, Germany, May 2008.

2 Biehl, J.T. Czerwinski, M. Smith, G. and Robertson, G.G. FASTDash: A visual dashboard for fostering awareness in software teams. In CHI, pages 1313–1322, SanJose, CA, USA, Apr. 2007.

3 Brackets, http://brackets.io.

4 Brown, J. Lindgaard, G. and Biddle, R. 2011. Collaborative Events and Shared Artefacts: Agile Interaction Designers and Developers Working Toward Common

Aims. In *Proceedings of the 2011 Agile Conference* (AGILE '11). IEEE Computer Society, Washington, DC, USA, 87-96.

5 Brun, Y. Holmes, R. Ernst, M. and Notkin, D. 2011. Proactive detection of collaboration conflicts. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 168-178.

6 Bugzilla. https://www.bugzilla.org.

7 Carzaniga, A. Rosenblum, D.S. and Wolf, A.L. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 2001. ACM Trans. Comp. Sys.

8 Carmel, E. Global Software Teams: Collaborating Across Borders and Time Zones. Prentice-Hall: Englewood Cliffs NJ, 1st edition edition, 1999.

9 Dourish, P. and Bellotti, V. Awareness and coordination in shared workspaces. In ACM Conference on Computer-Supported Cooperative Work, pages 107–114, Monterey, California, USA, 1992.

10 Ducheneaut, N. and Bellotti, V. E-mail as habitat: an exploration of embedded personal information management. Interactions, Volume 8(Issue 5): 30 – 38, 2001.

11 Ebert, C. and De Neve, P. Surviving global software development. IEEE Software, 18(2): 62–69, 2001.

12 Fitzpatrick, G. Kaplan, S. Mansfield, T. Arnold, D. and Segall, B. Supporting public availability and accessibility with elvin: Experiences and reflections. Computer Supported Cooperative Work, 2002.

13 Git. https://git-scm.com.

14 Goldman, M. 2011. Role-based interfaces for collaborative software development. In Proceedings of the 24th annual ACM symposium adjunct on User interface software and technology (UIST '11 Adjunct). ACM, New York, NY, USA, 23-26.

15 Grinter, R. Using a configuration management tool to coordinate software development. In CoOCS, pages 168–177, Milpitas, CA, USA, Aug. 1995.

16 Hancock, C. M. Real-time programming and the big ideas of computational literacy. PhD thesis, Massachusetts Institute of Technology, 2003.

17 Hattori, L. and Lanza, M. Syde: A tool for collaborative software development. In ICSE Tool Demo, pages 235– 238, Cape Town, South Africa, May 2010.

18 Herbsleb, J. and Moitra, D. Global software development. IEEE Software, 18(2): 16–20, 2001.

19 Herbsleb, J. Atkins, D.L. Boyer, B.G., Handel, M. and Finholt, T. A. Introducing instant messaging and chat in the workplace. In Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves, pages 171–178, Minneapolis, Minnesota, USA, 2002.

20 Herbsleb, J.D. Mockus, A. Finholt, T.A. and Grinter, R.E. Distance, dependencies, and delay in a global collaboration. In Proceedings of the 2000 ACM conference on Computer supported cooperative work, pages 319–328, Philadelphia, PA, 2000.

21 Kantor, M. and Redmiles, D. Creating an infrastructure for ubiquitous awareness. In Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001), pages 431–438, Tokyo, Japan, 2001.

22 Karolak, D.W. Global software development: managing virtual teams and environments. IEEE Computer Society, 1999.

23 Lvstrand, L. Being selectively aware with the khronika system. In Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW (91), pages 265–278, Amsterdam, NL, 1991. ACM Press, New York.

24 Mark, G. Extreme collaboration. Communications of the ACM, 45(6): 89–93, 2002.

25 McDirmid, S. (2013, October). Usable live programming. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (pp. 53-62). ACM.

26 McDirmid, S. Living it up with a live programming language. In Proc. of OOPSLA Onward!, pages 623–638, October 2007.

27 OMG. CORBACos: Notification Service Specification v1.0.1. 2002.

28 Perry, D. Siy, H. and Votta, L. Parallel changes in large-scale software development: an observational case study. ACM TOSEM, 10:308–337, July 2001.

29 Sarma, A. Bortis, G. and van der Hoek, A. Towards supporting awareness of indirect conflicts across software con- figuration management workspaces. In ASE, pages 94–103, Atlanta, GA, USA, Nov. 2007.

30 Sarma, A. Noroozi, Z. and van der Hoek, A. Palantír: raising awareness among configuration management workspaces. In ICSE, pages 444–454, Portland, OR, May 2003.

31 Sarma, A. A survey of collaborative tools in software devel- opment. Technical Report UCI-ISR-05-3, University of Cali- fornia, Irvine, Institute for Software Research, 2005.

32 Schneider, K.A. Gutwin, C. Penner, R. and Paquette, D. Mining a Software Developer's Local Interaction History. MSR 2004, Edinburgh, 2004.

33 Stefik, M. Bobrow, D.G. Foster, G. Lanning, S. and Tatar, D. 1987. WYSIWIS revised: early experiences with multiuser interfaces. ACM Trans. Inf. Syst. 5, 2 (April 1987), 147-167.

34 Tanimoto, S. VIVA: A visual language for image processing, J. Vis. Languages Computing, 127-139, June 1990.

35 Tigris.org. Subversion.

36 Van der Hoek, A. and et al. Continuous coordination: A new paradigm for collaborative software engineering tools. In Proceedings of Workshop on WoDISEE, Scotland, 2004.

37 Vessey, I. and Sravanapudi, A. P. Case tools as collaborative support technologies. Communications of the ACM, vol. 38:83–95, 1995.

38 Victor, B. Learnable programming. http://worrydream.com, Sept. 2012.

4 Conclusion

The purpose of our research was to improve collaboration and coordination in the context of numerous software related activities, from learning core concepts, to the design and implementation of large software products. We discussed how integrating collaborative coding and live programming supported by appropriate communication and user awareness can improve the quality of collaboration in two settings: Distance Learning and Design-centered Software Development.

We presented our tool Jimbo, a collaborative IDE with live preview. Then, we described two different editions of Jimbo: (1) Educational and (2) Professional. The former can be used to offer an active learning experience through remote pair programming and live coding to the students in the distance education, while the latter can improve the developer-designer relationship by offering synchronous collaboration and live programming, which allows designers to join the development process as first-class citizens. We then evaluated our claims in the both settings through multiple user studies.

First, we conducted a user study to show how remote pair programming can be as efficient as the traditional version in collocated settings. Then, we showed how the same ideas can be transferred to distance learning settings. Our results show that the integration of remote pair programming and live coding is beneficial and improves the quality of learning by bringing the students closer to their instructor as well as offering a fun and active learning experience.

Finally, we presented the results from a study in a design-centered software development setting and explained how the integration of collaborative coding; both synchronous and asynchronous; and live programming, supported by proper communication improved the collaboration and coordination between the designers and developers, enabling the designers to be more involved in the development process which led to a higher quality code and quicker development with less interruptions.

5 Bibliography

Al-Ani, B. Trainer, E. Ripley, R. Sarma, A. Hoek, A. and Redmiles, D. Continuous coordination within the context of cooperative and human aspects of soft- ware engineering. In *CHASE*, pages 1–4, Leipzig, Germany, May 2008.

Baheti, P., Gehringer, E. F., and Stotts, P. D. Exploring the efficacy of distributed pair programming. In Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Program- ming and Agile Methods - XP/Agile Universe 2002 (London, UK, UK, 2002), Springer-Verlag, pp. 208–220.

Berliner, B. CVS ii: Parallelizing software development. In USENIX Winter 1990 Technical Conference, pages 341–352, 1990.

Biehl, J.T. Czerwinski, M. Smith, G. and Robertson, G.G. FASTDash: A visual dashboard for fostering awareness in software teams. In CHI, pages 1313–1322, SanJose, CA, USA, Apr. 2007.

Brackets, http://brackets.io.

Brown, J. Lindgaard, G. and Biddle, R. 2011. Collaborative Events and Shared Artefacts: Agile Interaction Designers and Developers Working Toward Common Aims. In *Proceedings of the 2011 Agile Conference* (AGILE '11). IEEE Computer Society, Washington, DC, USA, 87-96.

Brun, Y. Holmes, R. Ernst, M. and Notkin, D. 2011. Proactive detection of collaboration conflicts. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 168-178.

Bugzilla. https://www.bugzilla.org.

Carmel, E. Global Software Teams: Collaborating Across Borders and Time Zones. Prentice-Hall: Englewood Cliffs NJ, 1st edition edition, 1999.

Carzaniga, A. Rosenblum, D.S. and Wolf, A.L. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems, 2001.

Cheng, L. Hupfer, S. Ross, S. and Patterson, J. Jazzing up eclipse with collaborative tools. In 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications / Eclipse Technology Exchange Workshop, pages 102–103, Anaheim, CA, 2003.

Cliburn, D., Experiences with pair programming at a small college, The Journal of Computing in Small Colleges, 19, (10), 20-29, 2003.

DeClue, T., Pair programming and pair trading: effects on learning and motivation in a CS2 course, The Journal of Computing in Small Colleges, 18 (5), 49-56, 2003.

Dewan, P. and Riedl, J. Toward computer-supported concurrent software engineering. IEEE Computer, 26(1): 17–27, 1993.

Dourish, P. and Bellotti, V. Awareness and coordination in shared workspaces. In ACM Conference on Computer-Supported Cooperative Work, pages 107–114, Monterey, California, USA, 1992.

Ducheneaut, N. and Bellotti, V. E-mail as habitat: an exploration of embedded personal information management. Interactions, Volume 8(Issue 5): 30 – 38, 2001.

Ebert, C. and De Neve, P. Surviving global software development. IEEE Software, 18(2): 62–69, 2001.

Ellis, C.A. Gibbs, S.J. and Rein, G.L. Design and use of a group editor. In Engineering for Human Computer Interaction, pages 13–25, Amsterdam, 1990.

Fitzpatrick, G. Kaplan, S. Mansfield, T. Arnold, D. and Segall, B. Supporting public availability and accessibility with elvin: Experiences and reflections. Computer Supported Cooperative Work, 2002.

Gaspar, A. Langevin, S. Active learning in introductory programming courses through student-led “live coding” and test-driven pair programming, EISTA 2007, Education and Information Systems, Technologies and Applications, July 12-15, Orlando, FL.

Git. https://git-scm.com.

Goldman, M. 2011. Role-based interfaces for collaborative software development. In Proceedings of the 24th annual ACM symposium adjunct on User interface software and technology (UIST '11 Adjunct). ACM, New York, NY, USA, 23-26.

Grinter, R. Using a configuration management tool to coordinate software development. In *CoOCS*, pages 168–177, Milpitas, CA, USA, Aug. 1995.

Hancock, C. M. Real-time programming and the big ideas of computational literacy. PhD thesis, Massachusetts Institute of Technology, 2003.

Hattori, L. and Lanza, M. Syde: A tool for collaborative software development. In *ICSE Tool Demo*, pages 235– 238, Cape Town, South Africa, May 2010.

Herbsleb, J. and Grinter, R.E. Splitting the organization and integrating the code: Conway's law revisited. In Proceedings of the 21st international conference on Software engineering, pages 85–95, 1999.

Herbsleb, J. and Moitra, D. Global software development. IEEE Software, 18(2): 16–20, 2001.

Herbsleb, J. Atkins, D.L. Boyer, B.G., Handel, M. and Finholt, T. A. Introducing instant messaging and chat in the workplace. In Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 171–178, Minneapolis, Minnesota, USA, 2002.

Herbsleb, J.D. Mockus, A. Finholt, T.A. and Grinter, R.E. Distance, dependencies, and delay in a global collaboration. In Proceedings of the 2000 ACM conference on Computer supported cooperative work, pages 319–328, Philadelphia, PA, 2000.

Horn, E. M., Collier, W. G., Oxford, J. A., Bond Jr, C. F., & Dansereau, D. F. Individual differences in dyadic cooperative learning. In Journal of Educational Psychology, 90(1), 153. (1998).

Kantor, M. and Redmiles, D. Creating an infrastructure for ubiquitous awareness. In Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001), pages 431–438, Tokyo, Japan, 2001.

Karolak, D.W. Global software development: managing virtual teams and environments. IEEE Computer Society, 1999.

Knister, M.J. and Prakash, A. Distedit: a distributed toolkit for supporting multiple group editors. In Proceedings of the 1990 ACM conference on Computer-supported cooperative work, Los Angeles, CA, 1990.

Krahn, R. Ingalls, D. Hirschfeld, R. Lincke, J. and Palacz, K. 2009. Lively Wiki a development environment for creating and sharing active web content. In *Proceedings of the 5th International Symposium on Wiki and Open Collaboration* (WikiSym '09). ACM, New York, NY, USA.

Lautamäki, J. Nieminen, A. Koskinen, J. Aho, T. Mikkonen, T. and Englundm, M. 2012. CoRED: browser-based Collaborative Real-time Editor for Java web applications. In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work (CSCW '12). ACM, New York, NY, USA, 1307-1316.

Lvstrand, L. Being selectively aware with the khronika system. In Proceedings of the European Conference on Computer Supported Cooperative Work, ECSCW (91), pages 265–278, Amsterdam, NL, 1991.

Mark, G. Extreme collaboration. Communications of the ACM, 45(6): 89–93, 2002.

McDirmid, S. Living it up with a live programming language. In Proc. of OOPSLA Onward pages 623–638, October 2007.

McDirmid, S. (2013, October). Usable live programming. In Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming & software (pp. 53-62).

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002, February). The effects of pair programming on performance in an introductory programming course. In ACM SIGCSE Bulletin (Vol. 34, No. 1, pp. 38-42).

McGuffin, L.J. and Olson, G. Shredit: A shared electronic workspace. Technical Report 45, Cognitive Science and Machine Intelligence Laboratory, Tech reports: 45, University of Michigan, Ann Arbor, 1992.

MOOCs on the Move: How Coursera Is Disrupting the Traditional Classroom (text and video). Knowledge @ Wharton. University of Pennsylvania. 7 November 2012. Retrieved 23 April 2013.

Murphy L., Blaha, K., VanDeGrift, T., Wolfman, S., Zander, C., Active and cooperative learning techniques for the computer science classroom, The Journal of Computing in Small Colleges, 18, (2), 92-94, 2002.

Nosek, J. T. ìThe case for collaborative programmingî, Communications of the ACM 41:3, March 1998, p. 105-108.

O'Donnell, A. M. and Dansereu, D. F. “Scripted Cooperation in Student Dyads: A Method for Analyzing and Enhancing Academic Learning and Performance,” in R. Hartz-Lazarowitz and N. Miller (Eds.) Interactions in Cooperative Groups: The Theoretical Anatomy of Group Learning, pp. 120-141, Cambridge University Press, 1992.

OMG. CORBACos: Notification Service Specification v1.0.1. 2002.

Perry, D. Siy, H. and Votta, L. Parallel changes in large-scale software development: an observational case study. ACM TOSEM, 10:308–337, July 2001.

Resig, J. Redefining the introduction to computer science. http://ejohn.org/blog/introducing-khan-cs/, 2012.

Rugarcia, A., Felder, R. M., Woods, D. R., & Stice, J. E. (2000). The future of engineering education. I. A vision for a new century. Chemical Engineering Education, 34(1), 16–25.

Sarma, A. A survey of collaborative tools in software devel- opment. Technical Report UCI-ISR-05-3, University of Cali- fornia, Irvine, Institute for Software Research, 2005.

Sarma, A. Bortis, G. and van der Hoek, A. Towards supporting awareness of indirect conflicts across software con- figuration management workspaces. In *ASE*, pages 94–103, Atlanta, GA, USA, Nov. 2007.

Sarma, A. Noroozi, Z. and van der Hoek, A. Palantír: raising awareness among configuration management workspaces. In *ICSE*, pages 444–454, May 2003.

Schneider, K.A. Gutwin, C. Penner, R. and Paquette, D. Mining a Software Developer's Local Interaction History. MSR 2004, Edinburgh, 2004.

Slavin, R. E. “Research on Cooperative Learning and Achievement: When We Know, What We Need to Know,” Contemporary Educational Psychology, 21, pages 43-69, 1996.

Stefik, M. Bobrow, D.G. Foster, G. Lanning, S. and Tatar, D. 1987. WYSIWIS revised: early experiences with multiuser interfaces. *ACM Trans. Inf. Syst.* 5, 2 (April 1987), 147-167.

Sun, C. and Ellis, C. Operational transformation in real-time group editors. In *Proc. Computer Supported Cooperative Work*, pages 59–68, 1998.

SunMicrosystems. Java Message Service API. 2003.

Tanimoto, S. VIVA: A visual language for image processing, J. Vis. Languages Computing, 127-139, June 1990.

Tigris.org. Subversion.

Van der Hoek, A. and et al. Continuous coordination: A new paradigm for collaborative software engineering tools. In Proceedings of Workshop on WoDISEE, Scotland, 2004.

Vessey, I. and Sravanapudi, A. P. Case tools as collaborative support technologies. Communications of the ACM, vol. 38:83–95, 1995.

Victor, B. Learnable programming. http://worrydream.com/LearnableProgramming, Sept. 2012.

Waldrop, M. Online learning: Campus 2.0 http://www.nature.com/news/online-learning-campus-2-0-1.12590

Warren, J. Rixner S. Greiner, J. and Wong, S. 2014. Facilitating human interaction in an online programming course. In *Proceedings of the 45th ACM technical symposium on Computer science education* (SIGCSE '14). ACM, New York, NY, USA, 665-670.

Werner, L. L. Hanks, B. and McDowell, C. 2004. Pair programming helps female computer science students. J. Educ. Resour. Comput. 4, 1, Article 4 (March 2004).

Williams, L. A., ìThe Collaborative Software Process PhD Dissertationî, Department of Computer Science, University of Utah. Salt Lake City, 2000.

Williams, L. and Kessler, R. R. “Experimenting with Industry’s ‘Pair-Programming’ Model in the Computer Science Classroom,” Journal on SW Engineering Education, Dec. 2000.

Williams, L. A. and Kessler, R. R. (2000, March). The effects of" pair-pressure" and" pair-learning" on software engineering education. In Software Engineering Education Training, 2000. Proceedings. 13th Conference on (pp. 59-65).

Wouters, P. van Nimwegen, C. van Oostendorp, H. and van der Spek, E. D. A meta-analysis of the cognitive and motivational effects of serious games. Journal of Educational Psychology, 105(2):249, 2013.