Collaboration Support for Novice Team Programming

Davor Čubranić Margaret-Anne D. Storey
Computer-Human Interaction and Software Engineering Laboratory (CHISEL)
University of Victoria, Canada
{cubranic, mstorey}@uvic.ca

ABSTRACT

Learning computer programming in a modern university course is rarely an individual activity; however, IDEs used in introductory programming classes do not support collaboration at a level appropriate for novices. The goal of our research is to make it easier for first-year students to experience working in a team in their programming assignments. Based on our previous work developing and evaluating IDEs for novice programmers, we have identified two main areas of required functionality: 1) features for code sharing and coordination; and 2) features to support communication. We have extended an existing teaching-oriented integrated development environment (called Gild) with features to support code sharing and coordination. We report on a preliminary study in which pairs of students used a prototype of our collaborative IDE to work on a programming assignment. The goals of this study were to evaluate the effectiveness and usability of the new features and to determine requirements for future communication support.

Categories and Subject Descriptors: D.2.6. [**Software Engineering**]: Programming Environments—Integrated environments;
H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—Computer-supported cooperative work;

General Terms: Human factors, Design

Keywords: teaching programming, Gild

1. INTRODUCTION

Learning computer programming in a modern university course is rarely an individual activity. In addition to traditional labs, where students can interact with their teaching assistants and labmates, students now have round-the-clock contact with potentially everyone in the class through on-line course forums. Previous research has shown the benefits of collaborative learning in introductory computer science classes through pair programming assignments [5], with significant improvements to students' experience and performance in the course. However, pair programming is not the only way for students to collaborate on their assignments, and its value as a pedagogical tool has not yet been unequivocally established. Furthermore, it is a technique that has to be taught to students for them to benefit from it fully, and given its relative lack of widespread adoption in industry, many instructors may themselves have had little practical experience with it. Lastly, pair programming has a drawback that both students in a team have to work on their assignment at the same time. This may not always be feasible for larger assignments that can take a significant effort over a week or more, and where students tend to switch between synchronous, side-by-side work, and asynchronous, at-home sessions.

In the software development industry, the dominant model of working in a team is for each developer to work mostly individually, coordinating work with their colleagues as necessary and using tools such as source code versioning systems to integrate the individual contributions. However, computer science students usually are not exposed to such a development model until their third or even fourth year. The goal of our research is to make it easier for first-year students to experience working in a team.

So far, we have focused on determining requirements based on our previous work developing and evaluating IDEs for novice programmers. Our requirements incorporate 1) features to support code sharing and coordination; and 2) features to support communication. We have extended an existing teaching-oriented integrated development environment (called Gild) with features to support code sharing and coordination.

Gild is an integrated development environment for teaching and learning programming that has been developed by our group at the University of Victoria [6]. Gild was designed to simplify and add pedagogical support to an existing "industrial-strength" IDE (Eclipse [1]) to make it more appropriate for novice programmers and their instructors. Since Gild's release in January 2004, it has been used in introductory classes at the University of Victoria, as well as at several universities outside of Canada. We collected feedback on its usefulness, ease of use, and desired new features through focus groups and course experience questionnaires. The initial experiences at the University of Victoria are very positive. One of the most requested features has been support for collaborative learning.

In the remainder of this paper, we first discuss the requirements for an IDE for team programming in a first-year computer science course and describe a current working prototype. We then present an exploratory study in which pairs of first-year students used our system to collaborate on a programming assignment in a range of conditions in which we explored the kinds of communication features necessary to support collaboration. We conclude with a discussion of further improvements that could be made to a development environment for supporting teams of novice programmers.

2. DEVELOPING REQUIREMENTS FOR A COLLABORATIVE IDE

In this section we summarize the requirements for code sharing, activity awareness, and communication that should be present in a teaching-oriented IDE that supports team programming. We base the requirements on the existing research and on our experiences

[1]Eclipse is developed largely by IBM but released as open-source software: `www.eclipse.org`

and feedback we have collected from computer science instructors and students. Some of these requirements are similar to those for an IDE used in industry. However, some of the standard solutions used in the software industry are not entirely appropriate in the educational setting.

2.1 Code sharing and coordination

Source versioning and configuration management systems are the main mechanism for code sharing and coordination in software development today [4]. There is a wide variety of systems available, but Concurrent Version System (CVS) is arguably the de-facto standard, especially for open-source software. It is typically supported in most IDEs out-of-the-box, including Eclipse, so it is a natural candidate to use in any team project.

However, CVS emphasizes the wrong set of features from those needed for group software development in an introductory programming class. CVS targets software development that takes place over extended periods of time and in which programmers work in relative isolation from each other. In the CVS model, developers only periodically merge their work into a common code repository, usually when a major development step has been completed. Consequently, CVS provides features for managing sequences of revisions of files, multiple development branches, and different release versions of the code. These features are unnecessarily complex for small programs and teams that consist of only two or three students. A model that versions the entire project as a single unit is appropriate for this task and would make it easier to understand and manage the project's version history.

Another significant issue is that a large part of the collaborative aspect of a programming exercise is lost in CVS because there is no indication of what the other teammate(s) are doing. It would be far better to have a more closely-coupled collaboration. For example, the IDE should indicate—in real-time, if possible—the parts of files that are currently being worked on by other team members and warn of possible conflicts as soon as they arise (e.g., [8]).

2.2 Communication for collaboration

As Churchill and Bly observed in their study of distance collaboration using virtual environments like MUDs, communication in collaborative tasks is usually *about* the artifacts that are the object of collaboration [2]. This causes great difficulties when, for example, collaborators discuss changes to a Word document in a chat window (or in a phone conference), while at the same time trying to view the actual document in a separate application so the discussion can be placed in its proper context.

This is an issue that should also be carefully considered for a collaborative IDE. In our case, we need to support both synchronous and asynchronous communication. We see text messages as the primary communication medium (both as real-time chats and as persistent messages, like email or online discussion forums). It should be possible to anchor these messages in code (e.g., as in Jazz [1]), but to make communication about artifacts even more expressive, we also want to allow marking up of the artifact, for example, highlighting parts of it and attaching a text comment to this mark-up. Finally, because a comment can relate to disjoint sections of an artifact (or multiple artifacts), the links from communication to artifacts (and vice-versa) should allow multiple end points. Such support could be extended to communication between instructors and students, such as feedback on students' assignments.

3. IMPLEMENTATION

We have extended Gild with a working implementation of the code-sharing feature described above. This prototype does not yet implement communication support because we wanted to explore the benefits and drawbacks of communication modalities as offered in common third-party tools, such as MSN IM.

The prototype is written as an Eclipse plug-in that extends the existing user interface in Gild with a conceptually simple front end to a source code versioning system. Students working in Gild can share their work with their teammates through two operations: by *uploading* the code changes contained in their workspace into a shared repository, and by *downloading* the most recent changes from the repository into their local workspace.

In effect, the "upload" command takes a snapshot of one's local changes in one's IDE workspace, and puts it into a shared repository from which other team members can "download" the changes to bring their workspaces in sync across the team. This conceptually simple model is actually more robust behind the scenes: our implementation of code sharing in Gild still uses CVS to version the individual files as they are uploaded to and downloaded from the repository. This means that, for instance, we can identify simultaneous changes to the same file and avoid two students' overwriting each other's work. Such conflicts are indicated with a warning, and the student can manually merge the two versions into his or her workspace and then upload the new version of the project back to the repository. Finally, our user interface includes a "diff" facility to allow the user to preview the effect of a download or upload operation on the local workspace and the repository, respectively. Fig. 1 shows a screenshot of Gild with the preview dialog during the upload operation.

4. EVALUATION

As we elaborated in Section 2, we believe that for an IDE to support collaboration effectively, in addition to code sharing it must also include communication and activity awareness features. To determine the type and the extent of needed features, we conducted an exploratory study in which pairs of students worked on a programming task using our prototype for writing and sharing code, and a set of common third-party communication tools. Our goal in this study was to test the effectiveness and usability of the code-sharing features, as well as investigate the type of communication support that would be most appropriate for integration into an IDE.

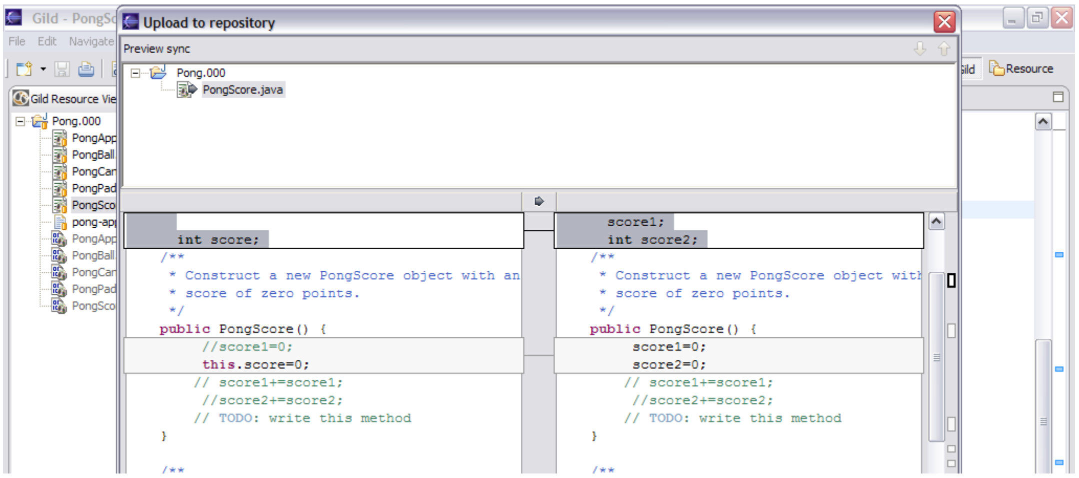
4.1 Design

Participants in the study were recruited from students enrolled in the second part of a two-course introductory computer science sequence usually taken in the first year. A total of ten students signed up, eight of them as pairs (P1–P4) and two individually (P5).

The participants in each pair were asked to collaborate on a programming assignment in which they were given a skeleton code for a computer game (Pong, see Figure 2) and had to implement the missing methods in four classes.[2] Three of the classes that had to be modified represented game objects—paddle, ball, and score—and the code that had to be written was either a "getter" returning the value of an object's internal state, such as position or current score, or simply had to update the object's state in response to game events. The fourth class was a subclass of Java canvas, and its `paint` method had to be completed by drawing the paddles and the ball in the right locations.

The pairs were randomly assigned to one of three conditions: face-to-face (two pairs: P1-F2F and P3-F2F), text-based instant messaging using MSN IM (also two pairs: P4-IM and P5-IM),

[2]The code for the task was based on Grant Braught's model assignment presented at CSE'03 [7] and available at `nifty.stanford.edu/2003/pong/`.



```
    int score;
  /**
   * Construct a new PongScore object with an
   * score of zero points.
   */
  public PongScore() {
      //score1=0;
      this.score=0;
     // score1+=score1;
      //score2+=score2;
     // TODO: write this method
  }

  /**
```

```
      score1;
      int score2;
  /**
   * Construct a new PongScore object with
   * score of zero points.
   */
  public PongScore() {
       score1=0;
       score2=0;
     // score1+=score1;
      //score2+=score2;
     // TODO: write this method
  }

  /**
```

Figure 1: A screenshot of Gild showing the preview dialog for the upload operation.

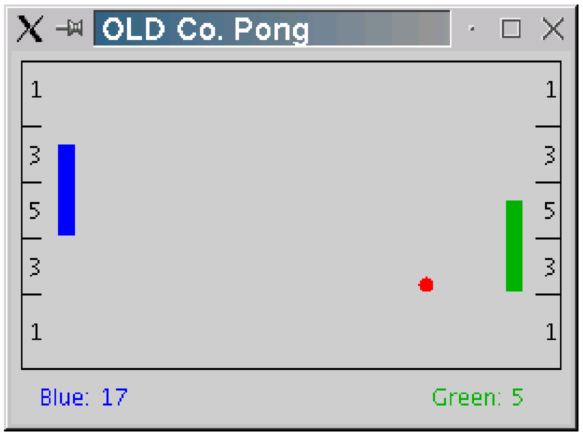


Figure 2: A screenshot of the Pong game used as the study task.

or audio- conferencing using Skype (one pair, P3-S). All groups started the session with both participants in a single room for the first 15 minutes, during which they could discuss the assigned task but not write any code. In non-F2F groups, one member was then moved to a different room, restricting the team to the respective computer-mediated communication (CMC) tool for the rest of the session. Pairs in all conditions had 45 minutes to work on their task, at which point we asked them to stop working and collected the code in their workspaces and the repository. The participants then responded to a multiple choice questionnaire on their experience in the assignment and took part individually in a brief interview in which they could describe their experience in their own words. During the study, we also videotaped the participants and recorded the contents of their screens using a screen capture program.

4.2 Results

As expected, there was a large variation in the students' programming ability and performance on the assignment. Only one of the pairs (P4-IM) finished the assignment in the time available, while others made varying degrees of progress. (The least progress, only one class completed and another started, was made by the other pair in the IM condition, P5-IM.)

The difficulties that the participants encountered, however, encouraged them to communicate extensively using the available tools, which was more important given that one of the purposes of this study was to determine the requirements for communication support. The participants reported that the support for code sharing that we added to Gild worked very well for their task. They liked the download preview capability, because it allowed them to view the changes their partner made to the repository prior to incorporating them into their own workspace. One participant, however, commented that he would prefer to be able to "backup" his workspace prior to downloading a change from the repository when the change was the wrong one to make. We note that an equivalent action would be to *back-out* of an uploaded change and restore the repository to an earlier version. This is a functionality that is already available in our back end, but is not currently accessible from the user interface. Based on this comment, it is something that we plan to address in future development.

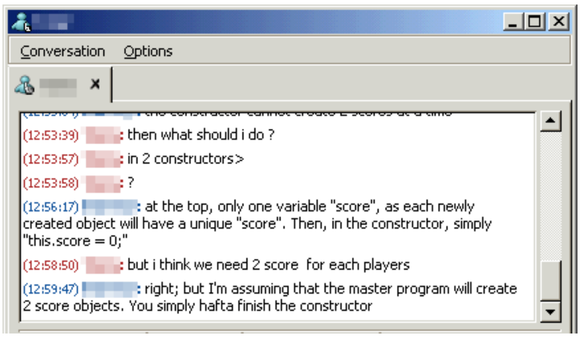
A more difficult challenge to address in our current code-sharing infrastructure is the lack of activity awareness, which is the result of using CVS internally for repository storage. This is not something that any participants noted as a problem, but the amount of explicit communication that took place about the status of uploads, such as "Have you uploaded your changes yet?" or "You can download now", is remarkable.

As we expected, communication between teammates was task- and artifact-centred. Participants in all conditions communicated when they needed help on the code or to monitor each other's activity. As one of the participants described it, the communication was limited to the immediate details of the code:

> [The communication was] just helping out with the details more, not—because it's difficult to describe exactly what the algorithm is, unless you upload it and everything, but mostly just for the details 'oh, how do I get this data where I need it, when I need it?'. (P2-S)

Although the participants had the first 15 minutes to discuss the organization of the skeleton code and their plan for implementing the assignment solution, they still occasionally needed to talk about how the game was supposed to work. This was especially the case for pairs that were unclear on the relationship of the game canvas class with the game model.

Furthermore, the communication usually referred to specific locations in the code, e.g., "you need to draw in the 'paint' method". When they needed to, the participants in the face-to-face condition



(12:53:39) [redacted]: then what should i do ?
(12:53:57) [redacted]: in 2 constructors>
(12:53:58) [redacted]: ?
(12:56:17) [redacted]: at the top, only one variable "score", as each newly created object will have a unique "score". Then, in the constructor, simply "this.score = 0;"
(12:58:50) [redacted]: but i think we need 2 score for each players
(12:59:47) [redacted]: right; but I'm assuming that the master program will create 2 score objects. You simply hafta finish the constructor

Figure 3: A screenshot taken during the study showing part of a conversation in a text chat (P5-IM).

would simply come over to their partner's computer to point at the screen when they needed to refer to a particular place in the code or its relationship with another. As previously observed by Churchill and Bly, participants in CMC conditions had to verbally describe the location about which they were communicating. Especially in the IM condition, this could be cumbersome, as can be seen in the conversation fragment shown in Figure 3.

In terms of satisfaction with the computer-mediated communication, the pair using Skype was happy with their experience, which was the first time they were exposed to Skype. Of the two pairs using IM, the P2-IM maintained frequent communication, almost every minute. As these students were regular IM users in their everyday life, they found this method of communication extremely easy. Both participants in the other IM pair, however, stated in their post-study questionnaire that they did not find using IM easy. We believe there are two reasons for this. First, one of the members of P5-IM pair was an infrequent user of IM-type applications. This was visible in this pair's communication patterns during the study, where all communication was initiated by the other participant. Second, the work in this pair was very asymmetrical: the participant who was always initiating the conversation had a lot of trouble with the code and needed his partner's help on every step. This help was difficult to provide effectively using only the text channel and the other participant found himself spending a lot of time away from his own work, which he somewhat resented.

4.3 Implications for future research

Based on our experiences and observations from this study, we plan to improve our code sharing feature with activity awareness indicators and a simple interface for reverting the repository to an earlier snapshot, for those situations when an uploaded change turns out to be the wrong approach.

Secondly, the study gave us somewhat mixed results on communication requirements for a collaborative IDE. While text messaging seems to be less effective than audio communication when one of the partner needs a lot of help with the assignment, it has two advantages. One is that snippets of example code can easily be typed in the chat window. The other is that non-native speakers may prefer it to speaking:

> It's more clearer to other people. It's better for me, I'm not first language, I speak second language, so I think it's harder for people to hear me. It's easy to just see what I write. (P1-F2F)

We are planning to investigate this issue further in a follow-up study later this summer, but expect that while both mediums should be supported, the solution we will likely adopt is incorporating instant messaging directly into the IDE (especially for reasons noted below) and continuing to use third-party tools for audio conferencing.

Also, with text chat, the transition from asynchronous to synchronous work can be done almost seamlessly if chats remain a permanent part of the workspace and team members can (re)join them as they come online and see the full contents of conversation so far (e.g., see[1]). In this study we have, for practical reasons, investigated essentially synchronous team work. Studying asynchronous collaboration over an assignment is part of our future plans, probably as an observational case study following a small number of students over a period of a full week.

Lastly, better support for linking the communication with the artifacts in the workspace is clearly necessary. There are multiple ways in which this could be realized. Text messages (or persistent chats) could be attached directly to the piece of code to which they refer, like in Anchored Conversations [3]. However, for synchronous collaboration, especially when using the audio channel, support for gestures becomes important, which will likely also require some form of screen sharing, such as that offered in VNC.[3]

5. CONCLUSION

We report on a preliminary study in which pairs of first-year computer science students used a prototype of our collaborative IDE to work on a programming assignment. The goals of this study were to evaluate the effectiveness and usability of the new code-sharing features we added as well as to determine requirements for future communication support. We found that code sharing through upload/download into and from a versioned repository was easy to use even for novice programmers. However, the lack of repository activity indicator meant that the participants had to compensate by verbally monitoring each other's progress.

Unlike more experienced programmers, our participants often turned to each other for help. Frequent references to specific locations in the code suggest that computer-mediated communication should be integrated into the IDE for easy remote gesturing and annotation of task artifact. Lastly, while it may be difficult to provide extensive explanation of code and programming concepts in text-based chat, non-native English speakers may prefer it for communication with native speakers.

6. REFERENCES

[1] L.-T. Cheng et al. Building collaboration into IDEs. *ACM Queue*, 1(9):40–50, 2004.

[2] E. F. Churchill and S. Bly. It's all in the words: Supporting work activities with lightweight tools. In *Proc. of GROUP'99*, pages 40–49, 1999.

[3] E. F. Churchill et al. Anchored Conversations: chatting in the context of a document. In *Proc. of CHI 2000*, pages 454–461.

[4] R. E. Grinter. Using a configuration management tool to coordinate software development. In *Conference on Organizational Computing Systems*, pages 168–177, 1995.

[5] C. McDowel et al. The impact of pair programming on student performance, perception, and persistence. In *Proc. of ICSE'03*, pages 602–607, 2003.

[6] D. Myers et al. Developing marking support within Eclipse. In *OOPSLA Eclipse Technology Exchange*, 2004.

[7] N. Parlante et al. Nifty assignments. In *Proc. of SIGCSE'03*, pages 353–354, 2003.

[8] A. Sarma et al. Palantír: Raising awareness among configuration management workspaces. In *Proc. of ICSE'03*, pages 444–454, 2003.

[3] `www.realvnc.com`