Real-Time Collaborative Coding in a Web IDE

Max Goldman, Greg Little, and Robert C. Miller
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
{maxg, glittle, rcm}@mit.edu

ABSTRACT

This paper describes *Collabode*, a web-based Java integrated development environment designed to support close, synchronous collaboration between programmers. We examine the problem of collaborative coding in the face of program compilation errors introduced by other users which make collaboration more difficult, and describe an algorithm for error-mediated integration of program code. Concurrent editors see the text of changes made by collaborators, but the errors reported in their view are based only on their own changes. Editors may run the program at any time, using only error-free edits supplied so far, and ignoring incomplete or otherwise error-generating changes. We evaluate this algorithm and interface on recorded data from previous pilot experiments with Collabode, and via a user study with student and professional programmers. We conclude that it offers appreciable benefits over naive continuous synchronization without regard to errors and over manual version control.

ACM Classification: D.2.6 [Software Engineering]: Programming Environments

General terms: Human Factors

Keywords: Collaboration, collaborative editing, pair programming

INTRODUCTION

In the current state of the art, software developers collaborate almost exclusively using one of two strategies: working together on a single copy of the code, using a single integrated development environment (IDE); or working in parallel on separate copies, integrating their efforts using a source code version control system. Neither approach, however, is ideal for close synchronous collaboration where both programmers actively contribute to the same piece of code. Using a single IDE (e.g. pair programming) means only the programmer who controls the keyboard and mouse can navigate, search, and write code. And using version control requires programmers to manually push and pull changes, while striving to avoid conflicts that require further manual resolution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

UIST'11, October 16-19, 2011, Santa Barbara, CA, USA.

Copyright 2011 ACM 978-1-4503-0716-1/11/10...$10.00.

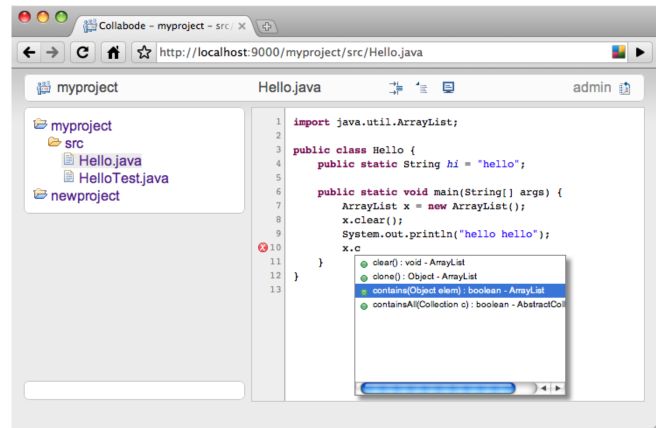

Figure 1: The Collabode web-based IDE allows multiple simultaneous editors to work together.

We have built a collaborative web-based IDE for Java, called *Collabode* (Figure 1), to study how a programming environment built to support close collaboration can improve the quality both of the collaboration and of the software produced [10]. In Collabode, changes by multiple programmers can be shared immediately, without the overhead of version control. This allows for more than one active contributor to the same module of code. Moreover, each programmer can use a different interface that supports their role in the collaborative effort, unlike the use of a single IDE where collaborators must share a single view.

Pilot studies with two to ten programmers working together simultaneously suggest that real-time shared editing can be effective and fun, and that it offers novel opportunities for collaborative software development [9]. However, a persistent problem has repeatedly bothered the collaborators: when one programmer introduces a bug, everyone has to live with that bug until someone fixes it. In particular, a compilation error created by one programmer's work-in-progress might prevent others from seeing less catastrophic compiler errors of their own, or might prevent them from running the program to test their code.

For example, suppose Alice and Bob are working together on the same Java class. Both are adding new methods, and Alice, working nearer the top of the file, begins (as any good programmer) by adding a comment to document her new method: she types /**, and suddenly Bob, working on his own method right below, is surprised to see his error markers

and syntax highlighting disappear. To the compiler, his code has temporarily become part of Alice's comment, until she resolves the incomplete syntax with `*/`. Such interference will be a constant drag on their collaboration.

In this paper, we examine an algorithm and user interface for addressing this issue of breaking the collaborative build, without introducing the latency and manual overhead of version control, and without unduly restricting the ability of both programmers to actively contribute. While programmers continuously and immediately see the text of one other's edits, the algorithm maintains separate working copies for each collaborator, compiling those copies separately to generate and display to each editor only their own errors. Once any subset of a programmer's edits can be applied to the on-disk version of the code without generating errors, those edits are applied and propagated to others' working copies.

In a study of how the algorithm behaves on recordings of collaborative programming sessions conducted without it, we found that it was able to integrate contributions that were originally copied manually into a master integration project. In another study where 14 student and professional programmers used Collabode with this synchronization algorithm to work in pairs on a realistic programming task, we found that developers were generally positive about the experience of collaborative coding and about the automatic error-mediated integration. Participants in the study were able to work without interference from their partner's unfinished code, and code was shared between participants approximately twice as often as it would have if participants had manually shared their code every single time it was error-free.

The contributions of this paper are (1) the Collabode web-based IDE for collaborative software development, (2) an algorithm for automatic error-aware integration that allows collaborating programmers to work in parallel, and (3) an evaluation of that algorithm as implemented in Collabode that demonstrates its efficacy for pairs of programmers working together in one source file.

After reviewing related work, we describe Collabode and the variety of collaborative programming scenarios it was designed to study; we discuss the algorithm, its implementation, and user interface; we describe and report the results of our evaluations; and then we conclude after some discussion.

RELATED WORK

Programming Methodologies

In *pair programming*, two developers work together on the same code in a single development environment, with the goals of improved communication and team knowledge sharing, increased productivity, and better software quality [2]. *Side-by-side programming* is a more flexible variant in which programmers sit together at separate machines [6]. This same style of interaction has also been studied with remote collaborators [7], and could be supported using a tool such as Collabode.

Collaborative Editing

Operational transform (reviewed in [25]) is the technique for maintaining a consistent view of the shared document that underlies most real-time collaborative editing systems, including Collabode.

Previous work in collaborative editing systems has examined the problem of maintaining semantic consistency, for example in graphic design [4] and hierarchical documents [15].

In the context of software development, *change impact analysis* computes the set of unit tests impacted by a collection of code changes [19] and was implemented in Chianti [18]; and *safe-commit analysis* determines code changes that can safely be committed to version control without breaking tests [29]. Most closely-related is the use of data dependencies and program slicing to perform *semantic interference detection* between changes made in parallel [24].

Development Tools

There exist a variety of commercial and open source systems for web-based collaborative programming. EtherPad enables real-time text editing collaboration and is used by Studio SketchPad (`sketchpad.cc`) for collaborative graphics programming. Ace (`ace.ajax.org`, formerly Skywriter, formerly Bespin), CodeMirror (`codemirror.net`), and Ymacs (`ymacs.org`) are web-based text editing components designed to be embedded in an IDE or other application. Kodingen (`kodingen.com`) is one such IDE for web programming, as are jsFiddle (`jsfiddle.net`) and CodeRun Studio (`coderun.com`). All three offer collaboration mediated by copying or version control – multiple programmers cannot edit the same files simultaneously. The Palm Ares environment (`ares.palm.com`) demonstrates an online graphical application development environment. Current research projects include Adinda [27], a web-based editor backed by Eclipse.

Flesce was one early implementation of a shared IDE to support authoring, testing, and debugging code [8]. The Jazz project [5] brought collaboration tools to Eclipse to support both awareness (e.g. via annotated avatar and project item icons) and joint work (e.g. with instant messaging and screen sharing). Different features of Jazz provide developer support throughout the software development process [26].

The CollabVS tool for ad-hoc collaboration shares similar awareness and collaboration goals [12], and the Sangam system was developed to support distributed pair programming [13]. Many other systems have focused on awareness features to keep loosely-collaborating software developers aware of others' work, e.g.: Palantír [22], Syde [11], Saros [21], CASI [23], and YooHoo [14].

COLLABODE: COLLABORATIVE CODING

Collabode is our web-based collaborative integrated development environment designed for experimenting with how programmers can write code in close collaboration with one another. Since the editor is a web application, programmers use a standard web browser to connect to a Collabode server that hosts their projects. The user interface is implemented in standard HTML and JavaScript and runs entirely in the browser.

New programmers can join a project and immediately start working together simply by visiting the same URL. There is no need to check out code or set up a local development

environment. This feature enables some of the new collaborative models discussed in the next section, and motivates the implementation of Collabode as a web application.

Collabode uses EtherPad (`etherpad.org`) to support collaboration between multiple simultaneous editors. Any number of programmers can open the same file simultaneously, and their concurrent changes are shared in near real-time to enable smooth collaboration whether they are working together remotely or at the same desk.

On the server side, Collabode uses Eclipse (`eclipse.org`) to manage projects and power standard IDE services: syntax highlighting, continuous compilation, compiler errors and warnings, code formatting and refactoring, and execution. The system also supports continuous test execution [20]. Collabode currently supports Java editing, and any Eclipse Java project can be compiled and modified using the system (including Collabode itself), with an interface familiar to anyone who has used Eclipse (Figure 1). Project code is executed on the server and clients can view console output, so Collabode is not currently suited for developing programs with desktop graphical user interfaces. As with source code, all collaborators can observe program output or open the console to view it after the fact, to support collaborative debugging.

The development of Collabode is ongoing work, and we are currently working on improvements to the user interface (especially project navigation and awareness of other editors) and expanding language support (console input, and supporting languages other than Java).

MODELS FOR CLOSE SYNCHRONOUS COLLABORATION

An immediate and obvious application of the Collabode editor is pair programming, whether remote or co-located, as well as the similar practice of side-by-side programming. Indeed, we believe that the pair or side-by-side programming experience in a collaborative system such as Collabode is substantially different from the experience of using a single editor, or two editors linked only via version control, and will offer some evidence of this in later sections.

A primary goal of this project, however, is to examine models of collaboration that go beyond pair and side-by-side programming. These models might take advantage of specific user interface support built into the collaborative development environment, and can allow programmers to take on distinct roles in the collaboration. We have previously proposed three interesting models to explore in this space: *test-driven pair programming* [10], *micro-outsourcing* [9], and *mobile instructor*.

In order to motivate our work on Collabode, we describe these models briefly below. In all of them, however, as well as in the straightforward pair or side-by-side programming model, and in the many other collaboration models one might imagine, experience piloting collaborative programming with both students and professional programmers leads us to anticipate a common problem: naive continuous synchronization of unfinished code makes it difficult for individual collaborators to make progress. Errors introduced by one programmer make it hard for others to understand the status of their own work, or to run or debug the project. Motivated by the exciting possibilities of the models below, the remainder of this paper will focus on that issue.

Test-Driven Pair Programming

This model combines the practices of pair programming with test-driven development. In test-driven development, developers follow two rules: "write new code only if an automated test has failed," and "eliminate duplication" [3], with short, rapid development cycles as a result.

In our test-driven pair programming model, the process of test-driven development is parallelized, with one member of the pair working primarily on tests, while the other works primarily on implementation. To begin work on a particular feature or module, the tester might write a black-box test, which the implementer will then satisfy. The tester can then investigate the implementation and write glass-box tests to weed out errors. These further tests will be addressed by the implementer, and the testing and implementing continues. Rapid collaboration in this model will rely on interface support for visualizing passing and failing test cases and who in the collaboration is responsible for taking the next step to address any failures, and for navigating between test cases and code.

Micro-Outsourcing

In this model, one programmer draws on the distributed expertise of a crowd of other programmers who make small contributions to the project. Micro-outsourcing allows the original programmer to remain "in the flow" at one level of abstraction or in one critical part of the code, while a crowd of assistants fill in the details or "glue" code elsewhere in the module or project.

In contrast to traditional outsourcing, which typically operates at the granularity of a whole module or project, micro-outsourcing requires a highly collaborative development environment and specific user interface support to make the collaboration effective. For workers, the interface must strike a balance between focusing on their limited task and allowing them to navigate and understand necessary dependencies or useful contributions from others. And for the original programmer, the system must make visible and understandable the small contributions of many others, allowing the programmer to easily integrate work or redirect the efforts of the contributors.

Mobile Instructor

Finally, we envision a Collabode-based system for students and instructors in a computer lab setting. Instructors and students work together to achieve educational goals, but their roles are highly asymmetric: the expert instructor structures the learning process, and the novice student participates in it. This asymmetry should be mirrored by a programming system designed for computer science education that goes beyond screen sharing tools such as iTALC (`italc.sf.net`).

While students continue to use the familiar IDE user interface, teachers utilize a mobile device interface that leaves them free to move around the classroom and work one-on-

one with students. This mobile view will summarize student progress and offer remote control of students' environments. Such a use case highlights the flexibility of a web-based IDE. Since the Collabode server has centralized up-to-the-second knowledge of every student's code, it can provide powerful tools to analyze or summarize student progress without interrupting their work or requiring student action.

COLLABORATIVE EDITING WITH SEMANTICS

Collaborative editing of source code files presents the particular problem that program errors introduced by one collaborator may disrupt the progress of their colleagues. Source files must conform to a particular programming language syntax and semantics, but it is impossible to expect users to maintain semantic validity throughout the coding process: almost any partially written expression, statement, block, method, class, or program is likely to contain errors according to the parser or compiler. Structured editors [17] address this problem only at the significant cost of requiring convoluted steps to make certain simple changes [28], and most programmers do not use them.

To see how errors introduced by one user can make it more difficult for another user to continue, consider the following simple program:

```
class Hello {
    int x;
    void a() { x = true; }
}
```

This contains an error, since the integer field `x` cannot be assigned a boolean value. However, suppose another user begins defining a new method:

```
class Hello {
    void b() {
    int x;
    void a() { x = true; }
}
```

At this point the Eclipse compiler will report instead that "`x` cannot be resolved to a variable," masking the true error because `x` now appears to be a local variable of method `b`. The particular set of failure cases and their likelihood and severity will depend on the compiler. Eclipse attempts to prevent common problems such as unbalanced braces from generating opaque errors elsewhere, but it cannot always succeed.

The problem is much worse in the case of running the program: although Eclipse will allow the programmer to launch an error-containing program, any attempt to execute an error-containing line will result in failure. This has the potential to prevent collaborating programmers from writing and testing their code incrementally, and we have observed this problem during pilot studies with both Python and Java programmers. Rather than constrain collaborating programmers to agree on and reach points at which the program is sufficiently error-free for them to run it and test code in progress, we instead account for the semantic content automatically in our synchronizing behavior.

```
1  public class Hello {
2      String a() { return "Alic
3      String b() { return "Bo
4  }
```

(a)

```
public class Hello {
    String a() { return "Alic

}
```

(b)

```
public class Hello {

    String b() { return "Bo
}
```

(c)

```
public class Hello {


}
```

(d)

Figure 2: Alice and Bob are working concurrently to define methods `a` and `b`. (a) The *union* of their changes is displayed in the editor, shown here from Bob's perspective: his un-integrated change is in yellow, with an outgoing arrow; Alice's is in gray, with an incoming arrow. (b) Alice's *working copy* and (c) Bob's working copy contain only their own changes. (d) The *disk* does not reflect their work-in-progress, since both of their changes still contain errors.

Overview

Collabode addresses this issue by first giving each programmer a separate, persistent *working copy* of the program, and then maintaining two versions that integrate programmers' changes from their working copies: a *disk* version and a *union* version. The union version is the text that users see and manipulate, and it contains all edits applied by all users, with highlighting and icons to indicate provenance. So if Alice has begun defining method `a` and Bob is writing method `b`, the union version contains both in-progress methods, and both methods appear in the user interface (Figure 2a). As long as their methods contain compilation errors, the working copies of Alice and Bob will each contain only their own method (2b-c) and the *disk* will contain neither (2d). Once their methods compile, their edits will be shared both with their collaborator's working copy, and with the *disk* version, which corresponds to the content on disk. It is this disk version that is run when either programmer elects to run the program. This version is always free of compilation errors.

Algorithm

We define an *edit* as the replacement of a single contiguous span of text with a new content string. Conventional additions and deletions are both special cases: an addition replaces a zero-length span with content, and a deletion replaces a span with zero-length content.

Given n users, the algorithm operates on several initially-identical (or empty) error-free documents $union$, $disk$, and wc_1, ..., wc_n. It also maintains the set E of edits that have not yet been applied to $disk$. This set is initially empty.

- Each new incoming edit e_i from editor i is applied both to $union$, for displaying the content of the edit to all editors,

and to wc_i, for computing the compilation errors to display to editor i.

- e_i is then merged with E. If e_i overlaps, adjoins, contains, or is contained by edits in E, the relevant (parts of those) edits will be removed.
- We then find a largest subset S of E such that when S are applied to $disk$, the resulting document is error-free. This set may be empty, and need not be a contiguous sequence.
- If S is nonempty, we apply each $s \in S$ to $disk$ and to all working copies where s was not already applied, and remove them from E.
- Finally, if it was changed, we write $disk$ to disk.

The result is that after each edit, we attempt to find a largest set of outstanding un-integrated edits that can be applied cleanly to the shared version on disk, and apply those edits if any. Since edits will frequently occur in rapid sequence, we need only merge them into E as they occur, and could delay subsequent searching for subset S until users are quiescent. We did not find it necessary to implement such a delay.

Implementation

The EtherPad component of Collabode implements simultaneous editing (using operational transform) so that other components, including the error-mediated integration, see only a sequence of edits as defined above. We have implemented the algorithm by further reducing incoming edits to contiguous *regions* of text that either exist only in $disk$, because they were deleted from $union$ but that deletion cannot yet be integrated without error; or exist only in $union$, because they were inserted but also cannot yet be integrated. The system maintains similar sets of regions correlating each working copy with $union$.

One limitation of this particular implementation is that the un-integrated edits in E from different authors must be non-overlapping; that is, if one author has inserted error-inducing text, no other editor can delete or modify that text. This limitation would become increasingly untenable as the number of authors in close proximity increases, but made it possible to implement and evaluate the idea more rapidly. It is not a limitation of the algorithm, only our current implementation.

More inherent to the approach is the limitation that, since deletions by one author are seen by all authors, they can no longer modify the text that was deleted, even if their working copy still contains that text because the deletion induces an error and is waiting in E. If we assume, however, that the authors have coordinated at some level regarding the deletion, in many cases there should be little need to modify code that they have already decided to remove.

A crucial assumption required by this algorithm is that the set E of edits not yet applied to disk is small, since we examine a number of possible combinations of those edits that grows exponentially. We evaluate this assumption below, but believe it is justified by two assumptions about programmer behavior:

- We assume that a programmer's own edits tend to be consecutive or overlapping. In this case, the edits can be merged and E remains small. Refactorings, which may require a large number of widely distributed small edits, break this assumption and would be better handled as single multi-location edits to be integrated whole.
- We assume that programmers tend to fulfill error-eliminating obligations actively rather than allowing large blocks of code to accrue with simple errors. Examples include balancing braces before working on their contents, and fixing minor syntax errors as they write rather than in a later proofreading step.

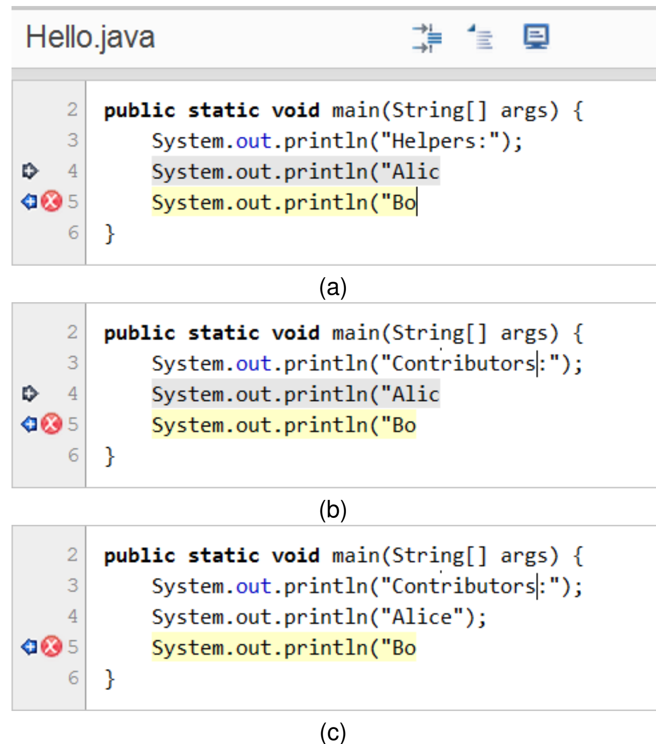

Figure 3: Bob's view of a collaboration with Alice: (a) `Hello.java` as both he and Alice begin adding to the `main` method; (b) after he improves the initial `println`, and his change is immediately integrated; (c) and after Alice completes her contribution, which is also automatically integrated.

Evidence for both patterns of editing exists in the case of Java programmers performing software maintenance [16]. The second assumption will be especially impacted by the details of compiler feedback. A tool such as DuctileJ [1], which allows programmers to temporarily relax static typing requirements and still run the program to see feedback, may reduce the scope of this assumption by giving the programmer flexibility to ignore what would otherwise be an error.

Interface

The result of applying this algorithm is shown in Figure 3. First Alice and then Bob begin making changes that do not compile (3a). Each of them sees only their own errors, and in-progress code is highlighted according to its origin. The figure shows Bob's view of their collaboration, so his un-integrated edits are highlighted in yellow, while Alice's are shown in gray. Arrows in the margin indicate Bob's 'outgoing' changes by pointing out of the text, while Alice's 'incoming' work is marked with inward-pointing arrows.

Figure 4: The output from running the program after Alice and Bob's collaboration: changes on lines 3 and 4 are integrated, but the unfinished line 5 exists only in Bob's working copy.

Once each of them completes edits that can be introduced to the project error-free (3b-c), those edits are applied to the version on disk. Then, even though one of Bob's changes still has errors, this does not affect either programmer's ability to run the project and debug their code (Figure 4). In this case, even though Bob began working first on line 5 and abandoned that work in progress to make a critical change on line 3, that completed change is not held back by his incomplete earlier work, and he needs to take no extra steps to share only the working contribution.

That the interface displays the work of both contributors is a crucial feature of the system. Each programmer has specific awareness about what other users are working on, and can shift attention to their code at any time. Both the clear benefits as well as some drawbacks to the interface were identified by participants in the user study discussed below.

EVALUATION: WILL IT BLEND?

In order to evaluate whether error-aware integration would be useful in practice, we first examined how the algorithm would perform in comparison to manual integration performed by programmers. We reviewed edit logs from pilot experiments conducted in Collabode without automatic integration and replayed them against a version of the system with the algorithm. In the original experiments, participants were research lab members engaged in *micro-outsourcing* as described previously. The original programmer (OP) directed co-located workers to complete various tasks (e.g. perform research, implement a method, design a representation), and then was required to manually copy the output of those workers into the original program.

Across three pilot sessions, we identified 18 instances where the OP copied worker code into the original project. In 14 of these instances (78%), the error-aware integration algorithm also would have automatically integrated the identical block of code into the project.

All four cases where the algorithm did not match manual integration were instances where the OP chose to integrate error-containing code. In two of those cases, the worker's contribution could reasonably be judged to satisfy the OP's specification: the OP requested that workers create a number of methods with correct signatures but empty bodies, yielding compilation errors in a number of places. In the other two cases, workers failed to meet the OP's specific instruction to "get rid of compile errors" in the initially broken code he provided them. It is therefore unclear whether the failure of the algorithm to integrate these contributions would have violated the OP's expectations.

The 78% of manually-integrated contributions that were also automatically integrated is a measure of the *recall* of this approach: what fraction of contributions judged "good" by humans are also selected algorithmically. We cannot use the same data to estimate the *precision* (that is, what fraction of algorithmically-selected contributions are "good") because in all of the pilot experiment sessions, there were large amounts of working code that OPs failed to integrate due to insufficient time or attention, not because the code was unacceptable.

This result gives us confidence that the algorithm can operate in a collaborative programming scenario without requiring programmers to change their behavior to meet its requirements. In fact, there were several instances where the OP incorrectly copied too little or too much code from the worker's project and had to correct the mistake. This effort would have been eliminated by the automatic approach.

EVALUATION: SIDE-BY-SIDE PROGRAMMING

To more fully evaluate the error-aware integration and the entire Collabode collaborative programming system, we conducted a user study in which pairs of programmers worked for approximately 30 to 40 minutes to complete a coding task, working together in a single source code file.

Four student participants were recruited from the authors' institution, and ten professional participants were recruited from local offices of two large technology companies. The average self-reported age of participants was 36, two of the 14 participants were female, and five were studying for or held a bachelor's degree while the other nine were studying for or held a master's degree. Participants were asked to rate their proficiency in Java on a 7-point Likert scale (min = 2, max = 7, mode = 6) and their experience with pair programming (min = 1, max = 5, mode = 1).

Pairs were *not* randomly assigned, and pair familiarity ranged from students who had never met prior to the study to coworkers who are also good friends. For this reason, and because of the wide variance in behavior and ability between programmers, we make no attempt here to compare the performance of different pairs; results are broken down by session only to illuminate this variance. This is also the reason we did not undertake a controlled experiment to compare Collabode to other collaborative editing approaches at this stage of the system's development.

In each one-hour session, participants:

- gave informed consent to participate and completed a demographic questionnaire;
- received a brief tour of the Collabode interface and the operation of the automatic integration algorithm, with the opportunity to ask questions;
- worked together on a programming task;
- completed a post-task questionnaire to give written feedback regarding their experience;

- and discussed the experience with the experimenter and their partner based on questions from the experimenter.

Tour

This experiment was not designed to test the learnability of Collabode, since it is by its nature a system for expert users who invest time to learn the complex tools of their trade. Participants were shown how to edit and run the project, and the experimenter demonstrated how multiple editors would see each other's changes. Participants took part in the demonstration but could not make edits, in order to keep idle experimentation to a minimum in the limited time.

Task

Participant pairs were first explicitly instructed 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 pairs used Google to conduct searches and referenced the Java API documentation at some point. Pairs were initially seated at a 90° angle at the corner of a rectangular table or at a small circular table, and the majority moved to sit side-by-side during the session. This move enabled easier sharing of documentation or other web pages, and also allowed those pairs to see one another's focus or cursor location (not currently a feature of Collabode). Pairs were also informed that they could only edit a single shared source code file; this restriction was designed to maximize opportunities to exercise the synchronization algorithm and UI.

The task was:

> Using the `nytimes.com` RSS feeds, display a list of recent headlines from the New York Times "Science" category. Use regular expressions or other string processing, not an RSS or XML processing library. Don't worry about converting XML entities or other encoding issues.

Participants were also given several hints in the initially-provided source code: pointers to the Times' directory of RSS feeds; to a class and a method relevant to retrieving the contents of a URL; and to the reference for Java regular expressions. While interesting, the research and API understanding stages were not the focus of this study, so we attempted to equip participants as well as possible to get to work quickly. Nevertheless, understanding the minutia of, variously, URL retrieval in Java, regular expressions in Java, and RSS consumed significant participant time.

Does the Set of Un-Integrated Regions Remain Small?

The efficiency of the automatic integration depends strongly on the assumption that there are few outstanding un-integrated regions. Considering all 4,543 edits performed in the course of the study, the average number of un-integrated regions for the algorithm to consider has a mean of 2.2 and a median of 2. The mode is 1, and since an edit has just occurred, the number is zero only when that edit precisely eliminates the only current region (this occurs 23 times). As shown in Table 1, pairs 6 and 7 tended to have about twice as many un-integrated regions, on average, as pairs 1 through 4 (up to the observed maximum of 15). Both of the these pairs experienced bugs in Collabode that prevented the shared code file from updating correctly, leading to invisible unresolved errors participants could not correct until they reverted to an earlier version.

These results validate our assumption that the set of such regions remains small, at least in the case of two programmers. Since we might expect this number to grow in proportion to the number of concurrent contributors, the efficiency of this algorithm beyond pair programming remains to be seen.

Is the Algorithm an Improvement Over Integrating Only Error-Free Code?

For each edit recorded during the study, we can ask whether that edit resulted (a) in an error-free source file and (b) in an automatic integration. Since sequences of edits where participants wrote inside comments or string literals result in rapid sequences of commits to the project that are not useful to count as separate instances, we collapse consecutive edits into blocks for which the answers to (a) and (b) remain constant.

In Table 1 we see a mean number of commit blocks per minute of 1.8, compared to a mean number of error-free blocks per minute of 0.7. This gives an average increase in the rate of commits by 2.8 times over a scheme that commits only when error-free (the result is an equivalent 2.6 times if we consider all edits without blocking). If we asked developers to integrate code manually using version control, this frequency of error-free states is likely an upper bound on the frequency with which they could reasonably go through the overhead of committing it.

A different way to view this result is to observe that 63% of commit blocks occurred with other error-containing regions still outstanding; that is, only some of the available edits were integrated into the project (this number falls to 58% when we examine edits individually, due to the effect of e.g. comments and strings noted above). We conclude that this algorithm is significantly different from integrating only error-free versions. More frequent integration means less opportunity for programmers' code to diverge and reduced lag between authorship by one programmer and use by another.

Can Developers Work Concurrently?

Table 1 also shows the proportion of edits attributed to the member of each pair who contributed the least. We see that in the most unequal case, that person still contributed more than one quarter of the edits (this is not necessarily a measure of how much code that participant wrote if, e.g., they copied an example from the web, or they tended to add but then delete). And on average, over one third of the edits occur in a state where both participants have an error, from which we might infer that they are actively working on or thinking about some part of the task. Based on experimenter observation, in every session, the two programmers were at some point actively working on incomplete code at the same time.

We argued earlier that, in addition to allowing participants to write independently and see appropriate IDE feedback, error-aware integration also allows programmers to run the program more often without worrying about their collaborators' errors. As shown in the last column of Table 1, the results from this study do not support such a hypothesis. Pairs ran

Session	Unintegrated regions	Commit block rate	Error-free block rate	Commit block ratio	Commit blocks with unintegrated regions	Edits by lower contributor	Edits where both editors have errors	Program runs with errors
1	1.6 per edit	3.0 per minute	1.3 per minute	2.3 per error-free block	59 %	41 %	14 %	0 %
2	1.7	2.3	1.0	2.3	58	38	20	20
3	1.6	0.9	0.7	1.4	52	43	41	0
4	1.4	1.1	0.8	1.4	49	38	19	0
5	2.4	2.3	0.5	5.1	80	48	65	0
6	3.2	1.5	0.4	3.6	72	31	43	100
7	3.2	1.3	0.4	3.3	72	26	50	25
Mean	**2.2** per edit	**1.8** per minute	**0.7** per minute	**2.8** per error-free block	**63** %	**38** %	**36** %	**21** %

Table 1: Per-session and mean statistics from the side-by-side programming user study.

the project only 5.7 times on average (4.7 times excluding consecutive runs with no intervening edits), and did so almost exclusively when the project had no outstanding error-causing code (with the exception of session 6). This is in contrast to statements made by participants during the post-task interview, which we discuss below.

Overall, participants made many edits where the errors reported were computed without regard to error-containing code concurrently introduced by their partner, and this offers strong support for our claim that developers are able to make concurrent progress using the system.

Feedback: Collaborative Coding

Study participants were largely positive about the experience of working together collaboratively, making statements such as "I found it really useful," "I thought it was a lot of fun," and "that was pretty cool."

In many instances, developers began to brainstorm different use cases for the system:

- "You could essentially be real-time debugging each other's code"
- "If you laid out an entire class with methods that needed to be implemented, [...] it would be pretty cool to see how that would work, if you were implementing one method, I was implementing another method" (cp. micro-outsourcing)
- From an operations engineer, "we have a lot of time-critical stuff, I could see this being occasionally useful" for "crisis programming"

Participants also suggested that the system could improve communication between collaborators, who can easily reference one another's code, and would be useful for learning. One participant said during the discussion, "I [...] gained an insight into how you [my partner] think, how you problem solve."

While pair programming works well for many developers, several participants reported that Collabode would work better for them than the traditional approach. One summarized the sentiments of many:

> "It always seemed that if you had one or two people looking over your shoulder, you became a puppet, and you just are doing what they're telling you to do. But when we each had a keyboard, it didn't feel that way. I was thinking one step ahead, and whether it was just going to the Java API documentation to see what's the next little thing that we can do to push this forward, [or working in parallel,] it was both of us trying to actually program this thing, instead of just one person."

Some participants were more ambivalent:

- "I think it boils down to, if you buy into pair programming, then this is a very good way to do it"
- "Once your task is well-defined, the module is well-defined, I wanted to go into a separate page"

One participant stated flatly, "overall, I wasn't a huge fan of it," and continued: "no, I don't like other people touching my stuff." While this is certainly a disappointing attitude from the perspective of collaborative software development, it is only one point along a spectrum of reasonable concern developers must have for their ability to think and code independently.

Feedback: Automatic Integration

Asked whether they thought the automatic error-aware integration was "useful" or perhaps "confusing," developers were generally positive:

- "I found it really useful, I liked the fact that you can see what they're doing, but that it doesn't interfere with the functionality"
- "I think that it's hugely useful"
- Without it, "I would have been totally hung up on, 'oh, what the heck is my stupid syntax error there,' and [...] that would have just stopped me in my tracks;" this participant's partner replied: "right, and we have an alternative, which was, you try your way, and I'll try my way," which they did during the session
- "I can picture myself spending ten minutes writing a method, [...] and if it doesn't compile, then [...] I don't want you to wait [on me], or me to wait on you"
- "I think you definitely don't want it to commit with errors, because then you can't test things as you're [...] making small changes that you want to be able to just test immediately"

- "I really like the idea that essentially whatever's in the system, whatever that is, will always run"

No developer reported being confused by the system as designed, although several reported being confused when bugs in Collabode were exposed. Nevertheless, some comments made by participants lead us to suspect that some participants had incorrect or incomplete mental models of the system, in particular with respect to what version of the program would run in the console.

Developers who thought error-mediated synchronization was less important tended to focus on running the program. One developer suggested that holding back changes with errors was unnecessary for a task of this size, although this participant's partner responded that they should have tried to run the code sooner, and that "we should have tested running the program with just my changes [and] when there was a compilation error with his changes, I should have tested running it." Another participant's statement that "I don't mind about the compilation errors because, when we cooperate, I think, 'I do not mind, if it's not ready, I just do not run"' also seems more closely aligned with the data presented above. We hypothesize that two factors combined to limit participants' use of this feature:

- First, the notion of a console that runs only the working parts of the code is novel, and participants may have preferred to synchronize rather than consider what would be run.
- Second, because the console output was shared, participants may have thought of running the program as a shared activity best done together when both were ready.

Feedback: Missing Pieces

Hitting on two of the limitations of the current system, several participants wanted to be able to see their partner's errors and to be able to edit their partner's un-integrated changes. One participant described how the incoming and outgoing change icons in the margin could provide an affordance for taking ownership of a change, which might help clarify for the change's originator why their code was now being edited. Several participants articulated how the ability to see collaborators' errors (perhaps with indicators to differentiate them from their own) was crucial for collaborative debugging. Some developers also mentioned wanting a changelog, and rightly pointed out that the current user interface becomes confusing with more than two collaborators because the specific owners of changes other than one's own are not displayed.

Several participants also made the case for when manual integration might be preferable in place of or in combination with automatic integration:

- "I like this as a default, but I think, it would be nice if you could hit a button that says, 'stop committing what I'm going to write,' [...] if I know I'm going to be working on a large chunk of code"
- "One place I see [manual] coming in really useful is, if you're working with someone, and you're pounding through the problem really quickly, and you solve it really inefficiently... and, say [...] I decided to go back and try to make some early part more efficient, I don't want to just delete the inefficient part, because then she can't test any more" if the intermediate state ever compiles

This is a particularly insightful observation from only half an hour of experience with the system: while error-aware integration is helpful, it might be straightforward to produce error-free programs that nevertheless impede the progress of collaborators. Improving the automatic integration algorithm and adding both language-specific and language-agnostic heuristics to detect "good" and "bad" changes is ongoing work.

DISCUSSION AND FUTURE WORK

Overall, we were pleased to hear from several participants statements such as "I want it" and questions about when the system would be production-ready for their own programming. Despite their enthusiasm, bugs in the concurrent interaction between the EtherPad shared editor and the automatic integration algorithm driving compilation in Eclipse caused Collabode to fail periodically, which frustrated programmers during the user study and often led them to treat the system warily rather than push its boundaries.

That participants in the study did not often run the program while it had errors means we can only speculate on whether having the program as it appears on disk be different than how it appears in the editor is a problem. Although they did not say so, users may have avoided running the program until it was error-free for precisely this reason. This question will require further evaluation.

Finally, it may be that resolving the issue of compiler errors impeding programmers' collaborative work will allow a different problem to dominate – changes that compile but produce runtime errors for other programmers. One approach we can apply to this problem is test-mediated integration, as in tools such as JUnitMX [30]. New collaboration structures and user interfaces may mitigate this problem without requiring test suites.

CONCLUSION

In this paper we have demonstrated the efficacy and usability of an algorithm and user interface for addressing the issue of errors in collaboratively-edited source code. We integrate only contributions that can be applied error-free, without introducing the latency and manual overhead of version control, and without restricting the ability of both programmers to actively contribute. This algorithm is implemented in Collabode, our browser-based collaborative programming system, which we are using to investigate how different models and user interfaces for close collaboration between programmers can yield novel and effective strategies for collaboration.

Why build a web-based IDE? Is this not merely a return to the days of terminals and mainframes, with VT100 replaced by shiny new HTML5 – now with animation! We believe otherwise. The collaborative opportunities offered by a browser-based IDE are too exciting to pass up: instant participation by anyone with a URL, collaboration with zero setup cost, and a development server that can serve appropriate user interfaces for collaborators using a variety of interfaces and devices.

ACKNOWLEDGEMENTS
We thank all the user study participants; Jay Goldman for helping to organize the study; and all the User Interface Design Group members who have contributed to this work, especially Patrick Yamane and Angela Chang. This work is supported in part by National Science Foundation award IIS-0447800 and by Quanta Computer as part of the T-Party project. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES
1. M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *ICSE*, page 521, 2011.
2. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
3. K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.
4. J. Bo, B. Jiajun, C. Chun, and W. Bo. Semantic consistency maintenance in collaborative graphics design systems. In *Proc. Computer Supported Cooperative Work in Design*, pages 35–40. IEEE, Apr. 2008.
5. L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up Eclipse with collaborative tools. In *OOPSLA workshop on eclipse technology eXchange*, 2003.
6. A. Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.
7. P. Dewan, P. Agarwal, G. Shroff, and R. Hegde. Distributed side-by-side programming. In *CHASE*, page 7, 2009.
8. P. Dewan and J. Riedl. Toward Computer-Supported Concurrent Software Engineering. *IEEE Computer*, 26:17–27, 1993.
9. M. Goldman, G. Little, and R. C. Miller. Collabode: Collaborative Coding in the Browser. In *CHASE*, page 65, May 2011.
10. M. Goldman and R. C. Miller. Test-Driven Roles for Pair Programmming. In *CHASE*, pages 515–516, 2010.
11. L. Hattori and M. Lanza. Syde: a tool for collaborative software development. In *ICSE*, pages 235–238, 2010.
12. R. Hegde and P. Dewan. Connecting Programming Environments to Support Ad-Hoc Collaboration. In *ASE*, pages 178–187. IEEE, Sept. 2008.
13. C.-W. Ho, S. Raha, E. Gehringer, and L. Williams. Sangam: a distributed pair programming plug-in for Eclipse. In *OOPSLA workshop on Eclipse Technology eXchange*, page 73, 2004.
14. R. Holmes and R. J. Walker. Customized awareness: recommending relevant external change events. In *ICSE*, pages 465–474, 2010.
15. C.-L. Ignat and M. C. Norrie. Handling Conflicts through Multi-level Editing in Peer-to-peer Environments. In *Proc. CSCW Workshop on Collaborative Editing Systems*, 2006.
16. A. J. Ko, H. H. Aung, and B. A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI Extended Abstracts*, CHI EA '05, page 1557, 2005.
17. S. Minor. Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, 37(4):399–418, Oct. 1992.
18. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *Proc. OOPSLA*, volume 39, 2004.
19. B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Workshop on Program Analysis for Software Tools and Engineering*, 2001.
20. D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *International Symposium on Software Reliability Engineering*, pages 281–292. IEEE, 2003.
21. S. Salinger, C. Oezbek, K. Beecher, and J. Schenk. Saros: an Eclipse plug-in for distributed party programming. In *CHASE*, pages 48–55, 2010.
22. A. Sarma, Z. Noroozi, and A. van Der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE*, pages 444–454, 2003.
23. F. Servant, J. A. Jones, and A. V. D. Hoek. CASI: preventing indirect conflicts through a live visualization. In *CHASE*, pages 39–46, 2010.
24. D. Shao, S. Khurshid, and D. E. Perry. Evaluation of Semantic Interference Detection in Parallel Changes: an Exploratory Experiment. In *International Conference on Software Maintenance*, pages 74–83. IEEE, Oct. 2007.
25. C. Sun and C. Ellis. Operational transformation in real-time group editors. In *Proc. Computer Supported Cooperative Work*, pages 59–68, 1998.
26. C. Treude and M.-A. Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *ICSE*, pages 365–374, 2010.
27. A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi. Adinda: a knowledgeable, browser-based IDE. In *ICSE*, pages 203–206, 2010.
28. R. C. Waters. Program editors should not abandon text oriented commands. *ACM SIGPLAN Notices*, 17(7):39, July 1982.
29. J. Wloka, B. Ryder, F. Tip, and X. Ren. Safe-commit analysis to facilitate team software development. In *International Conference on Software Engineering*, pages 507–517, 2009.
30. J. Wloka, B. G. Ryder, and F. Tip. JUnitMX - A change-aware unit testing tool. In *International Conference on Software Engineering*, pages 567–570, 2009.