MIT Libraries | DSpace@MIT

MIT Open Access Articles

Collabode: Collaborative Coding in the Browser

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Max Goldman, Greg Little, and Robert C. Miller. 2011. Collabode: collaborative coding in the browser. In Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '11). ACM, New York, NY, USA, 65-68.

As Published: http://dx.doi.org/10.1145/1984642.1984658

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/79662

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike 3.0

Massachusetts Institute of Technology

Collabode: Collaborative Coding in the Browser

Max Goldman
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
maxg@mit.edu

Greg Little
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
glittle@mit.edu

Robert C. Miller
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139
rcm@mit.edu

ABSTRACT

Collaborating programmers should use a development environment designed specifically for collaboration, not the same one designed for solo programmers with a few collaborative processes and tools tacked on. This paper describes *Collabode*, a web-based Java integrated development environment built to support close, synchronous collaboration between programmers. We discuss three collaboration models in which participants take on distinct roles: *micro-outsourcing* to combine small contributions from many assistants; *test-driven pair programming* for effective pairwise development; and a *mobile instructor* connected to the work of many students. In particular, we report very promising preliminary results using Collabode to support micro-outsourcing.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

Keywords

Collaboration, outsourcing

1. INTRODUCTION

Programmers constantly collaborate with one another, and with the current state of the art, collaboration is almost always mediated in one of two ways: by the shared use of a single computer and integrated development environment (IDE), or by the shared use of a source code version control system. Neither provides adequate support for close synchronous collaboration between programmers who actively contribute to the same module of code.

We have built an IDE, called *Collabode*, to study how a programming environment built to support specific structures of close collaboration can improve both the quality of the collaboration and the software produced. In Collabode, changes by multiple programmers can be shared immediately, and each programmer can use a different interface

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.
CHASE '11, May 21, 2011, Honolulu, Hawaii
Copyright 2011 ACM XXX XXX XXX ...$10.00.

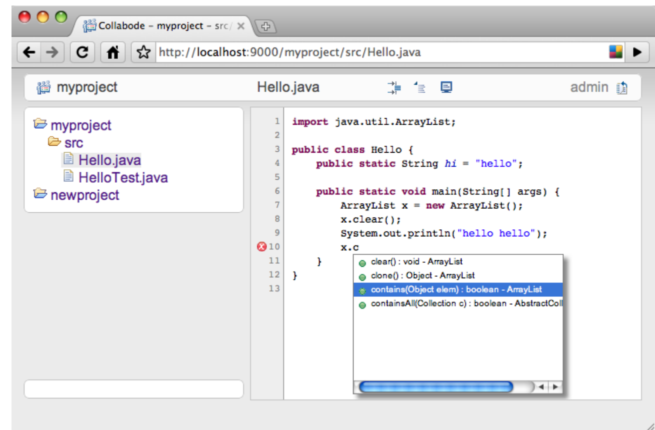

Figure 1: Collabode incorporates user interface features familiar to any Eclipse user, including code completion and highlighting errors.

that supports their role in the collaborative effort. This paper describes the system and reports on our own preliminary experience using it. We describe three novel collaboration scenarios: *micro-outsourcing*, where a developer is assisted by many others making small contributions; *test-driven pair programming*, a combination of side-by-side programming and test-driven development; and *mobile instructors*, where lab instructors use mobile devices to connect to the IDEs of their students. In each case, the web-based Collabode development environment can power user interfaces designed specifically for the collaborative structure and roles.

2. COLLABODE

We previously reported on the design and use of a Collabode prototype for Python programming [6]. Experience with that prototype has informed the construction of a Java development version designed to support realistic evaluation and potential deployment.

Collabode is a web application. Programmers use a standard web browser to connect to a Collabode server that hosts their project. The user interface is implemented in HTML and JavaScript and runs entirely within the browser. New programmers can join a project and immediately start working simply by visiting a URL; there is no need to check out code or set up a local development environment.

On the server side, Collabode uses Eclipse to manage projects and power standard IDE services: continuous com-

pilation, compiler errors and warnings, code formatting and refactoring, and execution. Any existing Eclipse Java project can be compiled and modified using the Collabode editor (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.

We use EtherPad (`etherpad.org`) to support collaboration between multiple simultaneous editors. An arbitrary 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. Collabode links EtherPad's synchronization to the Eclipse project file hierarchy, and details of synchronizing multiple programmers' work is described in subsequent sections.

Since Java programmers expect a variety of tool support, features such as syntax highlighting, code formatting, code completion, and library import organization are powered by Eclipse on the server and rendered with appropriate HTML- and JavaScript-based interfaces in the client.

3. MICRO-OUTSOURCING

Micro-outsourcing is the first of three novel models of collaborative programming we propose. In this model, a previously-solitary 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* (OP) 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 details and "glue" code elsewhere in the module or project. In contrast to traditional outsourcing, which typically operates at the granularity of a whole module or whole project, micro-outsourcing requires a highly collaborative development environment and specific user interface support to make the collaboration effective.

To explore how micro-outsourcing might be structured and to understand the barriers programmers will face, we are running sessions within our research group where students, professors, and visitors volunteer as OPs or crowd contributors. In this section, we discuss observations from five such sessions, each approximately 1 to 1.5 hours with five to eight participants.

These informal experiments suffer from several threats to validity. The OP feels a certain pressure to outsource as soon and as often as possible, knowing this is the point of the exercise. The crowd is not an anonymous pool of people, but rather a small group of colleagues waiting for the next chance to contribute. And both the eagerness and the skill level of these assistants likely overestimates the effort and experience offered by real-world workers, who might be recruited on an outsourcing marketplace like oDesk (`odesk.com`) or vWorker (`vworker.com`, formerly Rent a Coder). Nevertheless, clear trends still emerge.

3.1 Laziness, Impatience, and Hubris

Programming Perl (in addition to being something to avoid) offers three facetious "great virtues of a programmer: laziness, impatience, and hubris" [17]. Micro-outsourcing has been successful in our experience in part because programmers know how to use these virtues effectively.

Lazy programmers on the one hand write only as much as they need at the moment, but on the other hand write reusable code up front rather than re-write tedious routines again later. Lazy OPs are also happy to micro-outsource pieces of their code to others, and are able to navigate the tradeoff of spending energy to define and outsource tasks knowing their overall energy expenditure has decreased. In several of our sessions, both OPs and workers have remarked at the amount they could accomplish in little more than an hour, with very little per-task pressure.

Impatient programmers (according to [17]) write fast, effective programs that waste neither CPU nor the user's time. But in the context of collaboration with many others, impatience becomes the grease that keeps the wheels turning. Workers are impatient to complete their tasks and make a contribution, and the OP is impatient to see the results and build on them. Fast failure, when either workers or the OP abandon or rescind a task, is a feature we anticipate will make micro-outsourcing effective for rapid iterative development, and too-patient programmers might fail to fail.

Finally, hubris – the (excessive) pride that drives programmers to write great code – plays an important role. With micro-outsourcing, workers have continuous opportunities to prove their mastery of one idea, their skill with one construct, or their unmatched ability to answer one question. Just as programmers are proud to contribute excellent content to community wikis or sites such as Stack Overflow (`stackoverflow.com`), so we have observed in our experiments how contributors strive to do good work. And the OP can still play Zeus and strike down the work of unhelpful workers if necessary.

3.2 Breadth and Depth

Micro-outsourcing permits a wide variety of worker contributions: implementing a function or part of a class from a specification, writing test cases, searching for code examples, documentation, or libraries on the web, integrating a found code snippet into existing code, etc. All of these we have observed.

Asking workers to implement code, the original programmer sometimes leaves the specification implied ("implement iterator() please"), while other times might write detail in a Javadoc comment and point the worker there.

In one interesting case, the OP decided to implement a bag data structure. He began by asking a worker to outline the interface:

> Please write the method signatures for some operations I'm likely to need (use java.util.Set as a model): add, isEmpty, size, getCount(T element)

From two more workers, he requested: "Please choose a representation for this. I'm thinking a HashMap," resulting in two independent opinions on implementation strategy.

Workers also assisted with refactoring:

> Please change the bodies of the remaining methods (removeAll onward) so that they throw UnsupportedOperationException"

And with testing:

> Please write an @Test exercising Bag.add() and Bag.getCount(), using Bag<String>"

In general, we have observed a wide variety of code authoring and modification tasks, and tasks spanning from

API-related (“Find me the javadoc for [library] TagSoup”) to domain knowledge (“Teach me what straight flush means”).

3.3 Integrating Work

The original programmer usually begins by setting up a skeleton project ready to accept contributions. This takes the form of a class or classes with empty definitions or methods without bodies, which the programmer then requests workers to complete. The experience from there is dominated by the strategy used to synchronize code and integrate everyone’s contributions.

On one end of the spectrum, workers contribute in real-time to the OP’s original project. Everyone immediately sees everyone else’s work. This mode is most appropriate for situations where programmers must coordinate carefully, but it comes at a cost: making independent progress is more difficult because others have often broken the build. In particular, the OP has difficulty reaching a steady-state of cyclical development where they modify the code, test that it works, fix the bugs; and then add the next modification. Instead, the program remains ‘broken’ for a longer time.

At the other end of the spectrum is a clone-and-merge strategy, where workers are given clones of the project in which to make quiet, undisturbed progress. Upon completion of their task, the OP uses a live diff interface to investigate the changes and merge them back into the project. Naturally, this renders frequent outsourcing of small pieces of code – a major goal of micro-outsourcing – more difficult.

Between these two extremes, we are continuing to prototype new synchronization strategies. The key idea is to use signals for broken code – compilation errors, failing test cases – to determine whether a given contributor’s changes are ready for sharing or not, and if so, to share them automatically.

3.4 Worker Interface

Hired crowd workers are likely to put in less effort than the graduate students in our pilot experiments, so the question is how to build a contributor interface that encourages investment in the project. Collabode attempts to make contributing as easy as possible, with no setup required, so that workers can choose projects and individual tasks that interest them – if one enjoys writing regular expressions, rather than relish the one bit of regular expressions to be concocted in a large project, one can simply write only regular expressions for many projects.

An open question, however, is what sort of contextual information workers need: how much of the project do they need to see, read, or understand in order to work comfortably and correctly? Anecdotally, we find that the question is often turned on its head: workers are comfortable seeking out what they need, and are quick to recognize ambiguities or missing information in their tasks. If a worker makes a reasonable assumption and forges ahead, how can that assumption be communicated to the OP and to other workers? And if a worker needs clarification or assistance, what ought that interface to look like?

Since the competence and intentions of anonymous crowd workers cannot always be trusted, we have already implemented fine-grained read and write permissions on projects, packages, classes, and methods. Workers can be directed to a Collabode URL that will clone a project and grant them limited permissions to work in the cloned version, notifying

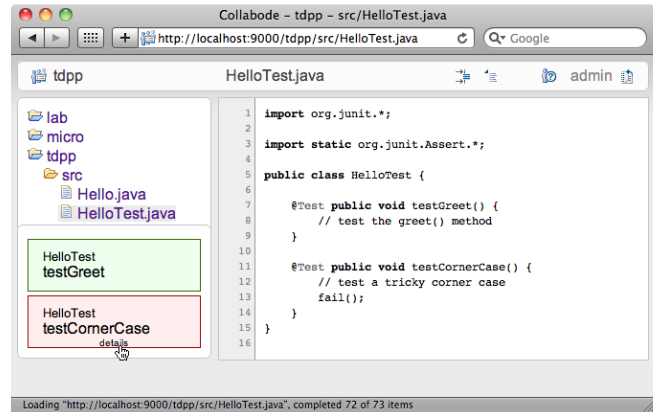

Figure 2: A project with one passing and one failing test case.

the OP when they have completed their task.

4. TEST-DRIVEN PAIR PROGRAMMING

We previously proposed *test-driven pair programming* as a model for close collaboration between programmers, combining pair programming and test-driven development, and reported on the results of a pilot study with the Python prototype of Collabode [6].

Pair programming is the practice of having two programmers 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 [1]. *Side-by-side programming* is a more flexible variant in which programmers sit together at separate machines [4]. And in *test-driven development*, developers follow two rules: “write new code only if an automated test has failed,” and “eliminate duplication” [2], 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 implementor, and the testing and implementing continues.

Addressing the main user interface needs identified from pilot study, Collabode implements *continuous testing* [11], where test cases are executed continuously and their status is reported to the programmers. Test cases are displayed so that both tester and implementer can easily see where they stand in the collaboration (Figure 2). Using code coverage analysis to link tests and implementation is ongoing work.

5. TO THE CLASSROOM AND BEYOND

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 experiences it. This asymmetry should be mirrored by a programming system designed for computer science education. Screen sharing tools, such as iTALC (`italc.sf.net`), provide some benefits, but the possibilities

afforded by a collaborative IDE are much greater.

We envision a Collabode-based system for students supervised by co-located *mobile instructors* in which students use the familiar IDE user interface, but teachers utilize a different view. In order to leave them free to walk around the classroom and work one-on-one with students, instructors use a mobile device interface that summarizes student progress, and offers remote control of students' development environments.

This 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. And because the Collabode client is web-based, we can easily craft new user interfaces for different collaboration modes, different situations, and different devices.

6. RELATED WORK

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. Mozilla Skywriter (`mozillalabs.com/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 [16], a web-based editor backed by Eclipse.

Flesce was one early implementation of a shared IDE to support authoring, testing, and debugging code [5]. The Jazz project [3] 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 [15].

The CollabVS tool for ad-hoc collaboration shares similar goals [8], and the Sangam system was developed to support distributed pair programming [9]. Many other systems have focused on awareness features to keep loosely-collaborating software developers aware of others' work, e.g.: Palantír [13], Syde [7], Saros [12], CASI [14], and YooHoo [10].

7. CONCLUSION AND FUTURE WORK

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 web browser, useful visualizations and graphical user interfaces that enhance collaboration, and a development server that trades local projects for analysis and tool support even when the user's access point is a phone.

Collabode is our attempt at a browser-based IDE for Java programming, combining Eclipse development tooling with real-time simultaneous collaboration. We are continuing to develop both the core system and the various collaborative interfaces described in this paper, and look forward to reporting the results of full-system experiments on the collaborative coding models here proposed.

8. REFERENCES

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[3] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up Eclipse with collaborative tools. In *OOPSLA workshop on eclipse technology eXchange*, 2003.

[4] A. Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.

[5] P. Dewan and J. Riedl. Toward Computer-Supported Concurrent Software Engineering. *IEEE Computer*, 26:17–27, 1993.

[6] M. Goldman and R. C. Miller. Test-Driven Roles for Pair Programmming. In *CHASE*, pages 515–516, 2010.

[7] L. Hattori and M. Lanza. Syde: a tool for collaborative software development. In *ICSE*, pages 235–238, 2010.

[8] R. Hegde and P. Dewan. Connecting Programming Environments to Support Ad-Hoc Collaboration. In *ASE*, pages 178–187. IEEE, Sept. 2008.

[9] 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.

[10] R. Holmes and R. J. Walker. Customized awareness: recommending relevant external change events. In *ICSE*, pages 465–474, 2010.

[11] 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.

[12] S. Salinger, C. Oezbek, K. Beecher, and J. Schenk. Saros: an Eclipse plug-in for distributed party programming. In *CHASE*, pages 48–55, 2010.

[13] A. Sarma, Z. Noroozi, and A. van Der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE*, pages 444–454. IEEE, 2003.

[14] F. Servant, J. A. Jones, and A. V. D. Hoek. CASI: preventing indirect conflicts through a live visualization. In *CHASE*, pages 39–46, 2010.

[15] 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.

[16] 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.

[17] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, 3rd edition, 2000.