Designing Collaborative Development Tools

Steven P. Reiss, Alexander Tarvo
Department of Computer Science
Brown University
Providence, RI. 02912 USA
{spr,alexta}@cs.brown.edu

***Abstract*—Collaboration between programmers can take various forms. A wide variety of tools have been developed to support these forms. As part of our development of the Code Bubbles integrated development environment, we have been attempting to design and develop tools that support collaboration within the working set context the environment provides.**

In this paper we briefly describe Code Bubbles, the various tools that we have developed to assist collaboration, and the types of collaboration each of these tools is designed to support. We conclude by describing the current state of the tools and our future plans.

***Keywords*—Programming environments, programmer communications, programmer logs, collaborative tools.**

I. Introduction

Software development today is largely a collaborative effort. Much software development is done by multiple programmers working in teams, often in conjunction with project managers, designers, users, and other stakeholders. However, there are many different styles and approaches to collaboration, each of which requires appropriate supporting tools.

At one extreme, very loose collaboration is required where a programmer is working on legacy code written by other developers. Programmers create copies of the code (by cloning a repository, downloading a tar ball, etc.) and make their own modifications or additions to it. Programmers have complete control over the code and do not necessarily plan to incorporate any changes into the original version. They might, however, want to communicate with the original developers to ask questions, report bugs, or provide suggestions.

A slightly more collaborative work style involves component-based collaboration. Here each programer works on a separate program components that interact with other components using "standard" interfaces. The interfaces are typically well-defined, but can (and often do) change over time. Communication among programmers is generally asynchronous via e-mail, shared web pages, or messaging.

A more collaborative work style occurs with repository-based collaboration. Here programmers share source through a clone-based source control system such as *git* or *svn*, possibly with file locking. This style requires additional communications via messaging, video conferencing or phone. Such collaboration is common in open source projects and in large companies where developers are separated geographically. Here each programmer maintains their own copy of the complete system source. Programmer communication generally uses a mixture of synchronous and asynchronous methods.

The tightest collaboration is required when multiple programmers work simultaneously on common source files. This occurs when developers are engaged in pair programming or if they use older source control systems such as *cvs* or *sccs* that work by locking individual files in a common working directory. Communications here is again a mixture of synchronous and asynchronous methods, but with the balance shifted more towards synchronous.

Each of these collaboration styles requires different development tools for communication, logging, and actual collaboration.

Communication tools let programmers exchange information while developing software. Tools for asynchronous communication include wikis, bulletin boards, FAQ lists, E-mail and text messages. Synchronous communication tools include instant messaging, phone calls, and video conferences.

Logging tools store knowledge about the software system, helping programmers become aware of what their colleagues are doing or have done. Source control systems provide basic logging functionality in form of check-in descriptions — notes made by programmers when they commit code. Check-in descriptors inform other programmers what has changed and who is responsible for change. Bug tracking databases provide more elaborate descriptions of work items in the software project including the rationale for changes. More sophisticated tools such as Mylyn [7] allow attaching external notes directly to the code including references to bugs or work items, and links to wikis or blogs that describe changes.

Collaborative tools facilitate programmers working simultaneously on the same code base. These tools usually work with source control systems, for example, by highlighting changes implemented by other developers [3] or providing assistance in merging changes [8]. They can also address problems involving concurrent work on common source, for example, providing a shared editor view such as that offered by Google documents.

Our interest is in extending today's development environments with appropriate tools to support this wide variety of different collaboration styles. In particular, we are interested in developing extensions to the Code Bubbles environment that adequately support collaboration.

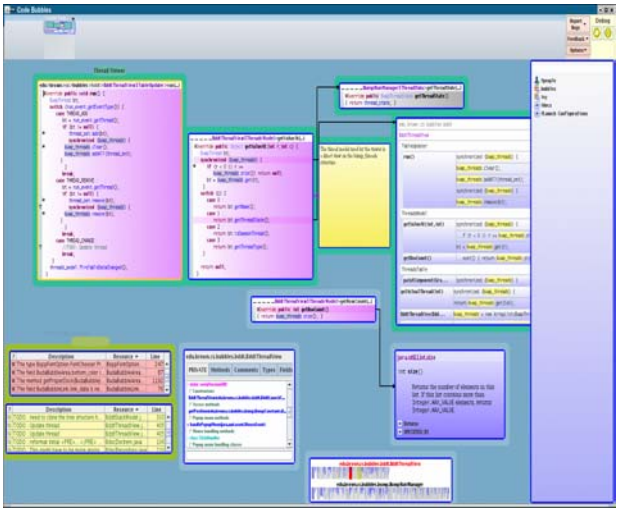

Fig. 1. Code Bubbles display showing a variety of different bubbles include source bubbles, notes, a bubbles stack showing search results, documentation, error messages, and context views.

II. Code Bubbles

Code Bubbles [1,2] is an attempt to redesign the user interface to programming to match the developer's working model. Code Bubbles represents a software project as a *working set*, a collection of *task-relevant fragments* including code, documentation, test cases, notes, bug reports, and other aspects of programming [9,10].

The fragments in a working set may be contained in multiple files, classes, or other modules, so quick and easy viewing of a working set is complicated in traditional environments. Code Bubbles presents fragments in fully manipulatable interface elements in order to provide an intuitive arrangement of the working set. Code bubbles is designed so that a working set should fit completely on a display. The working set is actually a viewport onto a much larger available work space. Multiple working sets can be scattered over the work space. A sample Code Bubbles display can be seen in Figure 1.

Code Bubbles provides a number of tools designed to help the programmer. It offers several techniques for code navigation including search facilities, fragment linking, and code overviews. It integrates tools for testing, code search, documentation, dynamic debugging, and education. We have also been working on developing a broad suite of tools to support the different collaborative styles.

III. Collaboration Tools In Code Bubbles

Collaboration support has been a concern throughout the development of Code Bubbles. Not only have we been developing tools for collaboration, but the underlying architecture of the system was designed with collaboration in mind.

Code Bubbles is built using a message-based client-server architecture. The back end (for Java, Eclipse with an additional plug-in to support the environment) maintains the working set. The actual Code Bubbles code acts as a front end that implements the user interface and the tools. The back end supports editing and updating of files simultaneously from the different front ends. It supports automatic notification of code changes, errors, and runs for all front ends. This enables multiple programmers to work on the same code simultaneously.

In addition, Code Bubbles offers a variety of collaboration tools. For communication it provides:

- A chat tool within the environment that can connect to any of the common chat services (Google, AOL, XMPP). The tool saves chat histories as part of the project for future reference. It also lets programmers send either images or loadable descriptions of their working sets as part of the chat.
- The ability to send e-mail messages from within the environment that include a description of the working set that can be reloaded in the recipients environment. These descriptions can also be saved into files that can be sent separately, included as part of a wiki or blog, or saved for later use.

Code Bubbles provides its own set of logging tools. These include:

- A note tool that allows programmers to create bubbles containing HTML-based notes. These notes are saved as part of the working set and are automatically reloaded when the working set is reloaded. Each note can be attached to a line of code without actually modifying the source file. In this case the note becomes a permanent part of the programmer's project and is accessible by other programmers viewing the code.
- A flag tool that allows the insertion of flag icons into working sets to alert other programmers of particular conditions.
- A programmer's log facility that automatically records programmer actions such as looking or editing a particular method. This is similar to facilities provided by Mylyn [6], Synde [5], or SpyWare [11]. Our tool has the potential to provide more useful information since the environment works at the method level rather than the file level and working sets typically correspond to individual tasks. Our tools automatically prompts the programmer to identify what task they are working on and lets the programmer easily add notes, images, and working set dumps to the log. Logs can be shared among all the programmers in a project. Logs can also be queried selectively by task, file, date, or programmer. A view of the programmer's log can be seen in Figure 2.

Code Bubbles offers several tools for supporting collaborative development. These include:

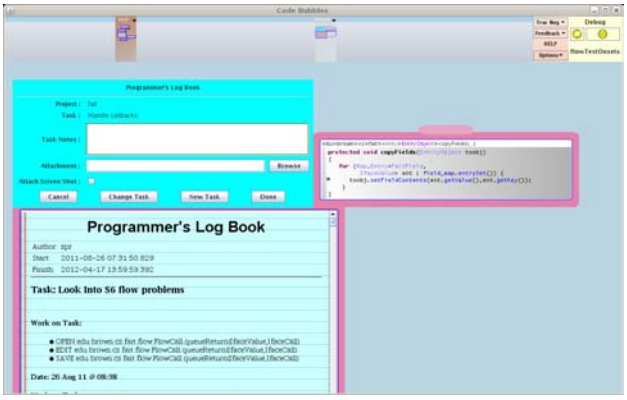

Fig. 2. A view of the programmer's log showing both the prompt window and a log display.

- A source-control package that integrates with common source control systems including *svn*, *git*, and *cvs*. This package automatically detects the source control system used for the current project and lets the rest of the Code Bubbles query and access source control information.
- A conflict detection tool that informs the programmer if the code he is currently viewing has been modified by other developers since he checked it out. This alerts programmers to potential conflicts during a future check-in.

  Whenever source files are changed by the developer, the conflict detection tool computes the differences between the current files and the last checked out version. It creates a file describing these differences and sends that file to a central server. When a developer opens a new file, the tool retrieves all relevant difference files from the server and computes the set of source lines changed by other users relative to the version the developer is working on. Changed lines are marked in the editor by a colored band in the annotation bar and by associated tool tips. This provides immediate feedback to the developer that the method they are looking at has been or is in the process of being changed by other developers.

  Similar facilities are offered by Crystal [3] or the IBM's Rational Team Server (Jazz). Unlike these tools, conflict detection in Code Bubbles ensures secure communication and does not require complicated setup. The programmer only needs to add a private UID file to the top level of the source project. This UID is used to encode all the files being sent through the central server. Encryption ensures the server will not have access to the original code or other sensitive information. Code Bubbles also support private servers where needed.
- A tool that provides shared display regions corresponding to working sets. A user can mark a working set as shared. Any other Code Bubbles interface working with the same back end can then display the shared working set. Any new bubbles, bubble movements, etc. done in any instance of the shared working set are sent to and duplicated in all the other instances. This provides programmers with the ability to do remote pair programming or to support interactive code reviews.

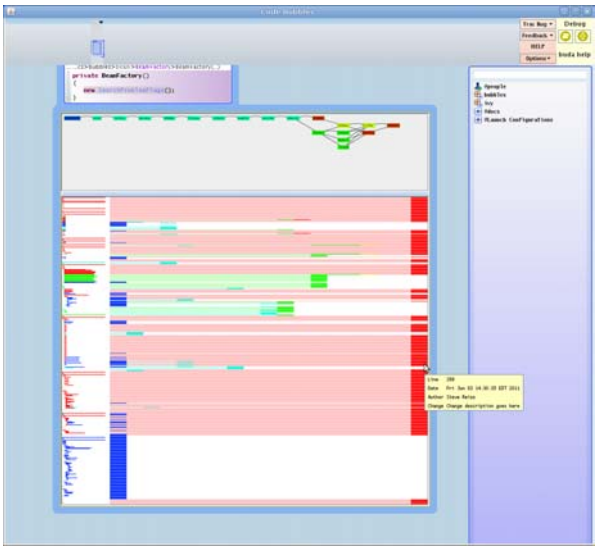

Fig. 3. Preliminary version of the Code Bubbles file history tool. The top of the tool shows which checked-in versions of the system modified the file and the branching structure of these versions. The bottom display shows where each line was created and changed over time. Colors correspond to different authors.

IV. Current Work

While we have developed a number of collaboration tools in Code Bubbles, we have not yet determined if these tools are appropriate, adequate, or sufficient. We are currently working on evaluating the effectiveness of the existing collaboration tools. We hope that such evaluations will help us to improve existing tools and develop new ones.

In particular, we are looking at extending the note tool to allow embedded audio and video notes in the working set. Similarly, we are considering adding a video chat to the set of the communication tools.

We are working on mining project information from software repositories and presenting it to the programmer in a compact and effective manner. Although such information is widely used for educating new developers [4] and for early failure detection [12], there was little effort on incorporating it into the IDE.

One possible use of such mined information would be helping the programmer to understand the prior evolution of the code he is working on. For example, when the programmer opens a method in Code Bubbles, he should have the option of seeing the history of prior commits for the method. This would show what parts of the code are volatile or new, would identify the developers with expert knowledge of a particular part of the system, and could identify other portions of the project with similar change histories. A preliminary version of such a tool showing file history by line colored by developer is shown in Figure 3.

Additionally, the mined information would allow computing important code metrics and incorporating them into the Code Bubble interface. For example, frequently changed methods could be considered to be more fault-prone and highlighted with different color. Another potential use of mined information would be discovering relations between different fragments in the working set. In particular, developers might want to know which functions were affected by the particular bug, or what parts of the system could be affected by the change. Code Bubbles could represent such information in form of interconnected bubbles.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation grant CCF1130822. Additional support has come from Microsoft and Google.

REFERENCES

1. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *ICSE 2010*, pp. 455-464 (2010).

2. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," *CHI 2010*, pp. 2503-2512 (2010).

3. Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin, "Crystal: precise and unobtrusive conflict warnings," *Proc. 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering*, pp. 444-447 (2011).

4. D. Cubranic and G. C. Murphy, "Hipikat: recommending pertinant software development artifacts," *Proc. ICSE 2003*, pp. 408-418 (May 2003).

5. L. Hattori and M. Lanza, "An environment for synchronous software development," *ICSE Companion*, pp. 223-226 (2009).

6. Mik Kersten and Gail C. Murphy, "Using task context to improve programmer productivity," *Proc. SIGSOFT 06/FSE 14*, pp. 1-11 (November 2006).

7. M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," *Proc. Aspect Oriented Software Development '05*, pp. 159-168 (2005).

8. Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce, "A formal investigation of Diff3," *Foundations of Software Technology and Theoretical Computer Science* (*FSTTCS*), (December 2007).

9. Andrew J. Ko, Htet Aung, and Brad A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," *Proceedings of the 27th ICSE*, pp. 126-135 (2005).

10. B. A. Meyers, A. J. Ko, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. on Software Engineering* Vol. **32**(12) pp. 971-987 (2006).

11. R. Robbes and M. Lanza, "Spyware: a change-aware development toolset," *Proc. ICSE 2008*, pp. 847-850 (2008).

12. A. Tarvo, "Mining software history to improve software maintenance quality: a case study," *IEEE Software* Vol. **26**(1) pp. 34-40 (2009).