A Research Demonstration of Code Bubbles

Andrew Bragdon[1], Steven P. Reiss[1], Robert Zeleznik[1], Suman Karumuri[1], William Cheung[1], Joshua Kaplan[1], Christopher Coleman[1], Ferdi Adeputra[1], Joseph J. LaViola Jr.[2]

[1]Brown University
Department of Computer Science
{acb, spr, bcz, suman, jak2, wcheung, cjc3, fadeputr}@cs.brown.edu

[2]University of Central Florida
School of EECS
jjl@eecs.ucf.edu

ABSTRACT
Today's integrated development environments (IDEs) are hampered by their dependence on files and file-based editing. We propose a novel user interface that is based on collections of lightweight editable fragments, called bubbles, which when grouped together form concurrently visible working sets. We describe the design of a prototype IDE user interface for Java based on working sets.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments.*

General Terms
Human Factors

Keywords
Integrated development environments, concurrent views, working set, source code, bubbles, navigation, debugging, human factors.

1. INTRODUCTION
Programmers spend between 60-90% of their time reading and navigating code and other data sources [1]. Programmers form working sets of one or more fragments corresponding to places of interest [2]; with larger code bases, these fragments are scattered across multiple methods in multiple classes. Viewing these fragments concurrently is likely to be beneficial, as it has been shown that concurrent views should be used for tasks in which visual comparisons must be made between parts that have greater complexity than can be held in limited working memory [3].

Because contemporary integrated development environments (IDEs) are file-based it is difficult to create and maintain a view in which multiple fragments are visible concurrently. This requires the programmer to manually and repeatedly perform numerous interactions to place, resize, scroll, and wrap long lines in a different file window for each fragment. Instead, IDEs are optimized for switching among different views using tabs, forward/back buttons, etc. Perhaps as a result, programmers may spend on average 35% of their time in IDEs actively navigating among working set fragments [2], since they can only easily see one or two fragments at a time.

In this paper, we argue in favor of a new approach, where the IDE shows multiple editable fragments concurrently, letting the user see and work with complete working sets. The result reduces navigations and supports new higher-level interactions over and within the working set.

Our approach is founded on the metaphor of a *bubble* – a fully editable and interactive view of a fragment such as a function, method documentation, or debugging display. Bubbles, in contrast to windows, have minimal border decoration, avoid clipping their contents by using automatic code reflow and elision, and do not overlap but instead push each other out of the way automatically. Bubbles exist in a large 2-D virtual space where a cluster of bubbles comprises a concurrently visible working set.

Code Bubbles is presented in [7] [8], and is more fully-described there, along with several user studies which evaluate its efficacy. In this paper we focus on providing a summary of the system.

ICSE '10, May 2-8, 2010, Cape Town, South Africa

2. RELATED WORK
User interfaces for classical programming languages have a long history. The work closest to the bubbles approach let the programmer work in terms of program fragments. These efforts let the programmer edit in terms of individual functions, or similar units. This was the approach we took in Desert [9] [10] and it can also be seen in IBM's Visual Age environments [11] and in the Sheets environment from CMU [12]. All these were loosely based on non-file based programming languages such as Xerox's Smalltalk and its successors, various versions of Lisp, and visual languages such as NI's LabView. Another approach is that of JASPER which displays small read-only views that represent the user's current task as a means for navigation [13]. A number of tools have been developed to add navigation aids to file-based environments, e.g. Mylar [14]. These navigation tools focus on identifying working sets, whereas we focus on displaying working sets concurrently.

3. THE BUBBLES METAPHOR
The basis for the user interface of our IDE is the bubble metaphor described fully in [7]; in this section we will briefly recap the metaphor and then in the next section present the extensions we made to design a prototype IDE user interface built on this metaphor. The bubbles metaphor represents working set code fragments (typically functions) as individual bubbles (Figure 1-L) that can be freely positioned on the 2-D display surface (1-Q). In addition, the display surface is treated as a portal on a large scrollable canvas which both lets more bubbles be open in the workspace than fit onscreen and also encourages programmers to pan over (thus preserving their working set views) to create room for new working set fragments when needed. This metaphor fundamentally differs from the multi-window UI used in some IDEs, such as Visual Studio or Eclipse, because it addresses four critical problems associated with window displays:

- Code does not naturally fit into arbitrarily sized windows
- Viewing overlapping windows requires manual interaction
- Window decorations are distracting and space consuming

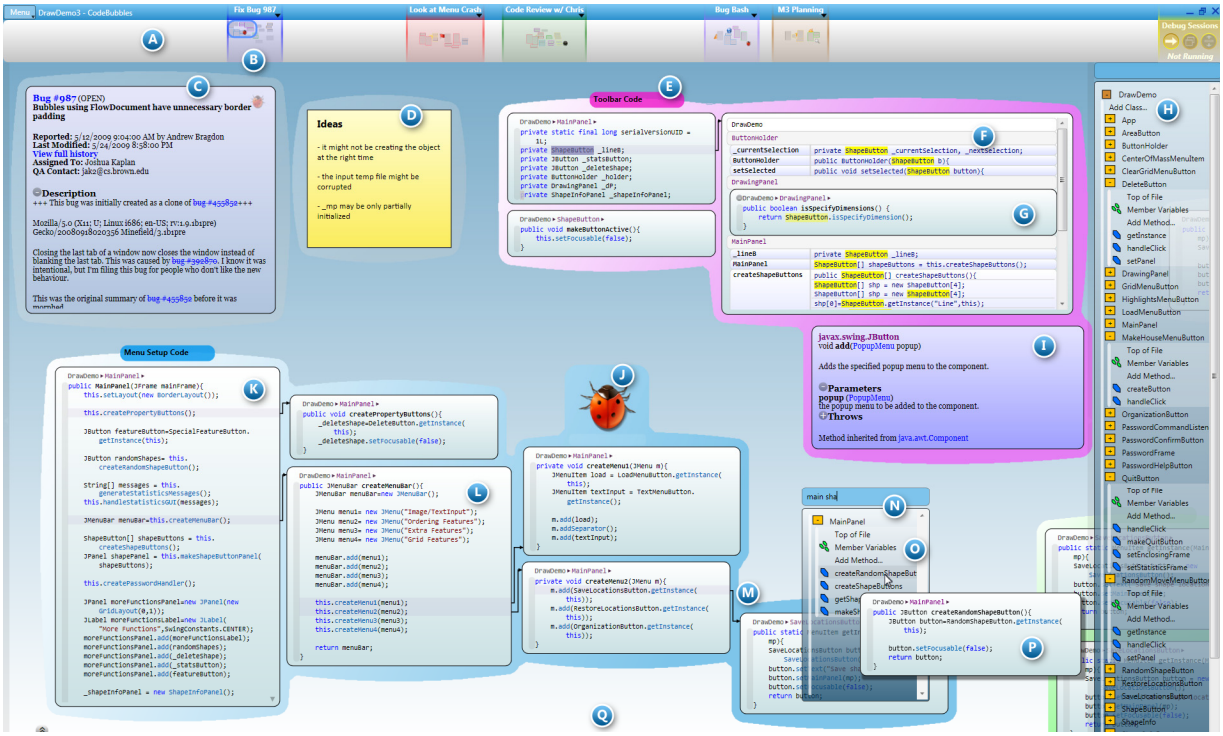

Figure 1.
The Code Bubbles IDE. Resolution: 1920x1200 (space reserved for task-bar).

- Eventually, the user will run out of space as he/she works on a series of tasks

To ensure that code can be easily read and edited regardless of the dimensions of its bubble, bubbles never clip text horizontally, but instead automatically reflow long lines. This approach produces similar results to those generated manually by programmers when splitting long lines. Additionally, bubbles vertically elide lengthy functions by default at the block level, and support subsequent user-based expansion. Reflow and elision are only applied to the view; they do not edit the underlying text.

Bubbles are also not allowed to overlap each other, making groups of bubbles easier to read since no Z-order management is needed. When one bubble is moved on top of another, a bubble spacer automatically moves the overlapped bubbles out of the way using a simple, recursive, heuristic algorithm that minimizes the total movement of bubbles.

To facilitate the simultaneous display of large numbers of bubbles, bubbles have no space-consuming UI decorations (i.e., scroll-bar, title-bar, etc) other than a thin border line and a bread-crumb bar (see top of 1-L). Instead, programmers interact with bubbles using right, middle, and left buttons respectively to move, close or edit text within bubbles. In addition, the scroll wheel is used to scroll text and dragging the left-mouse button on a bubble border initiates resizing. The breadcrumb bar provides the bubble's context by displaying the package and class name. Clicking on the class or package name provides direct access to peer methods and variables via a drop down list.

Background annotations are also used to highlight important inter-bubble relationships. For example, when Open Definition is chosen for a method call, a rectilinear arrow connection (1-M) is added to indicate the calling relationship between the resulting method definition bubble and the bubble containing the call.

Bubble stacks (1-F) are used by commands which logically return sets of bubbles, such as Find All References. Bubble stacks present results in two columns, the first listing the function containing the result, and the second showing the line in question with the result highlighted. Results are grouped by package, class and method. Clicking an item expands it in-place as a bubble (1-G). Since each such command results in a new bubble stack, users can easily compare results side-by-side.

4. IDE USER INTERFACE

Building on the bubbles metaphor as a foundation, we have developed a functional IDE user interface that includes many of the features of traditional IDEs, and novel features that fundamentally leverage the bubbles approach. These new features are centered on working sets. Some techniques make it easy to create displays of useful working sets, while others use displays of working sets to provide direct access to functionality that would otherwise be unavailable or cumbersome.

4.1 Compatibility Techniques

Since not all techniques benefit directly from the bubbles metaphor, we extended our interface to include standard tools. We display a docked package explorer pane (1-H) on the right-hand side of the display for exploring and adding new classes, methods and imports. We also pop-up a pane with compiler errors, docked to the bottom of the display, when they occur. We provide keyboard shortcuts for common functionality, including the ability to change keyboard focus between bubbles, bring up the popup search box, pan, and zoom.

Within bubbles, developers can edit code in much the same way they do with a conventional editor. If needed, they can open a full class in a bubble, perhaps to initially enter the code for an entire class at once. Developers can also "bud" a new method from an existing bubble; when users insert a new line at the bottom of a method in the desired class, and begin typing the method's declaration it will split off into a new bubble that grows as the user types, pushing bubbles below it out of the way using the bubble spacer. We also provide several menu-based methods for adding classes and methods. We implemented traditional auto-complete, and augmented it with a working set-oriented technique; programmers can type a new method signature not in the list and create an empty bubble for the new method to be filled in later.

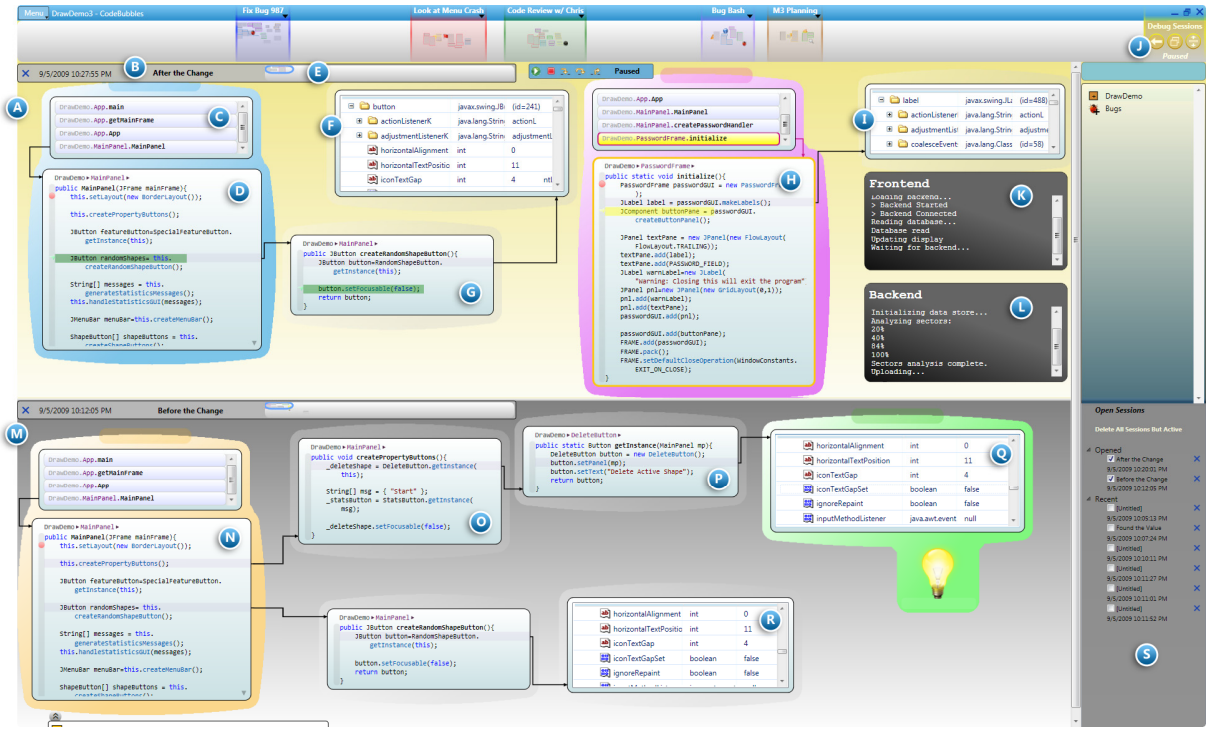

Figure 2.
Debugging with the Code Bubbles IDE. See Debugging, above.

4.2 Building Heterogeneous Working Sets

To create a bubble display for any method in the full package hierarchy, a popup search box (1-N) can be displayed by right clicking on the background. Using Boolean substring matching, programmers need only type brief fragments of a class or method name to rapidly filter the list of matched methods and open a bubble from the list. Hovering over a result shows a preview (1-P).

While reading and editing source code is important, we realize that much of what a programmer does within an IDE goes beyond code. We thus provide specialized bubbles that let users create richer task-relevant working sets.

Javadoc bubbles (1-I) let users browse through the documentation for a class, field or method. Javadoc bubbles provide appropriate elision controls and popup search box integration makes it easy to find the appropriate documentation with minimal data entry. *Note bubbles* (1-D) let users add formatted text annotations as sticky notes sharable with others. *Flag bubbles* (1-J) are a lightweight means of associating an icon and optional label with code and are useful for annotating bugs and to-do items, for creating hyperlinks, and for generally creating visual markers. *Web bubbles* provide access to a simple but full-featured web browser within the bubble framework. *Bug bubbles* (1-C) provide a bubble view of bugs from a bug tracking database based on Bugzilla.

The developer can display call paths by drawing a connecting line between two functions open in bubbles with the mouse; the backend performs a static call graph analysis to determine if there is a path in the call graph between the two functions. If more than one path exists, the shortest path is used. New bubbles and connections are opened for each function in the path, and are inserted between the two existing bubbles.

4.3 Lightweight, Persistent Groups

In addition to individual bubbles, our front end supports *bubble groups* (1-E) - which provide a simple means for defining and saving working sets and tagging functions. Groups automatically form when bubbles are brought close enough together; they are displayed using a common background color for the group, can be named using a title box, and are supported by extensions to the bubble spacer. These extensions both support group membership and provide an interface for splitting groups.

Groups persist automatically. They can then be reloaded on demand. They can be used as the target of a search, based on a substring match of group name and/or contents. They can also be the basis for a search, letting the user see bubbles that are related to a particular bubble by means of saved group membership.

4.4 Interruption Recovery and Multi-tasking

The *workspace bar* (1-A) at the top of the display is an extension of the simple panning bar from our previous work [7] that supports the definition of working sets for a particular task or goal.

The workspace bar operates by extending the screen space in the X and Y directions and provides access to different areas of that space by simply clicking in the bar. The bar provides a high-level overview map of the bubbles throughout the virtual display. *Sections* of the bar (1-B) can be labeled for task management. The bar and its sections are continuous rather than discrete so that these sections can be easily extended to occupy more or less space incrementally as a task grows or shrinks in size. The map is detailed enough to show the icons associated with flag bubbles.

The workspace bar provides a simple means for handling interruptions. If an interruption requires working on the project in a different way, the programmer can easily move to a different area of the virtual space, do the new work in that space, and then simply move back to where they were when they were interrupted. Task naming can help keep track of the interrupted and new tasks.

While the task bar is quite large, it is not infinite. To support prolonged development, we allow the user to close and save sections of the task bar for later use. These sections appear in a list we call the *task shelf* where they are displayed with their name and date. The user can reload closed task sections by clicking.

4.5 Debugging with Bubbles

One of the most important functions of an IDE is to aid the programmer in debugging. While we wanted to use bubbles to provide convenient access to traditional debugger support, the

lightweight nature of bubbles and the ability to have significant numbers of them displayed at once let us provide a much richer experience by showing program context over time.

Traditional debuggers provide displays of the program state at a single point in time. However, programmers often need to understand what changes over time, and to compare program state, data structures, etc. at the current time with their values at a previous time. Programmers may also want to annotate the program state with appropriate notes, observations, and ideas and to share this information with others.

Traditional debugger support is provided by a breakpoint bar to the left of the code, by toolbar commands or keyboard shortcuts for starting, stepping, continuing, and terminating an execution, and by allocating a section of the workspace bar for debugging.

When a program stops at a breakpoint or an exception, the user is taken to a new area of the debugging workspace (2-J), a code bubble is opened (2-D) for the code where the program stopped, and a bubble stack is opened to display the call stack (2-C). This bubble lets the user open methods from the call stack. If the user then steps into another method, a new code bubble is created (2-G) to the right of the current bubble and the bubbles are linked with a connector. Run time exceptions that stop the program also create exception bubbles displaying the Javadoc for the thrown exception. New bubbles opened in the debugger push bubbles that are siblings in the call hierarchy out of the way using the spacer.

Stepping out does not explicitly remove the prior function bubble. If the user next steps into another function, a new bubble is created to the right and below the prior call bubble. If the program stops in a new context (e.g., breakpoint hit), this context is placed to the right (2-H) of the prior one and the display is automatically panned. The result of this is a viewable history of the programmer's debugging actions displayed, where appropriate, as a tree.

Right clicking on a variable brings up a data structure bubble (2-I) showing the type, name and values of the selected object. These bubbles can be further expanded to show nested values. Typically, these bubbles are updated dynamically as the program executes. However, the user can either freeze a display, or they may "tear out" a subtree of the data structure and save the display for later comparison. Data structure bubbles for functions that are not being executed are saved for future comparison (2-F). In addition to a standard console, we support multiple virtual console bubbles (2-K, L); users can direct program output to particular consoles based on a user-configurable line prefix.

Each instance of a program being debugged is stored in a horizontal layout we call a *channel* (2-A). The system preserves views of previous debugging sessions (2-M) for comparison. Similar to the main workspace, each channel can be panned independently and has a miniature panning bar (2-E), providing a scrollable overview of the session. The panning bar lets the channel scale to accommodate a large or long session. Each session channel is accompanied by a title bar (2-B) that includes the modification date and an optional title. Sessions can be saved and reloaded using an interface (2-S) equivalent to the task shelf.

4.6 Sharing Information

The configuration of code or debugging bubbles along with appropriate annotations and flags provides a visual display of information relevant to the programmer, effectively a visual explanation. This can be printed, exported as PDF or saved for documentation or future use. Moreover, the saved configurations can be shared with other developers by simply e-mailing (using the built-in email-as-attachment option) the saved file and having them reload the bubble configuration in their own workspace.

5. LIMITATIONS

The prototype implementation of Code Bubbles is limited in several ways. It is resource-intense, requiring a modern dual-core CPU or better, a hardware-accelerated graphics card, and either one large (24") or two smaller (19") monitors to be effective. Large numbers of bubbles tend to degrade display performance. Many features one might expect in a complete IDE are missing: support for programming languages other than Java, portability, GUI and HTML designers, unit testing, XML files, database designers, and performance monitoring. The Code Bubbles editor is not as sophisticated as modern program editors such as Eclipse's. Working set definitions are not robust across significant external edits; this problem could be ameliorated by storing workspace information using file offsets, and applying the techniques developed in [15]. The debugging interface is currently optimized for problems in which an error in part of the call tree manifests immediately in the same branch of the tree. One advantage that files have over working with code fragments is that they can provide a readable and long-lasting context for programmers who need to read an entire class.

6. ACKNOWLEDGMENTS

The authors wish to thank Andries van Dam and Ken Hinckley for their advice and insight, and Donnie Kendall, David Eichler, Salman Cheema, Jared Bott, Jeff Coady and Max Salvas for their assistance. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship and in part by NSF grants CCR-0613162, and IIS-0812382.

7. REFERENCES

[1] Erlikh, L. Leveraging Legacy System Dollars for E-Business. *IT Pro*, May/June (2000), 17-23.

[2] Ko, A. J., Myers, B. A. et al. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE TSE*, 32, 12 (December 2006), 971-987.

[3] Plumlee, M. D. and Ware, C. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. *ACM Transactions on Computer-Human Interaction*, 13, 2 (June 2006), 179-209.

[4] Murphy, G. C. et al. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23, 4 (July/August 2006), 76-83.

[5] Robillard, M. P., Coelho, W., and Murphy, G. C. How effective developers investiage source code: An exploratory study. *IEEE Trans. on Software Engineering*, 30, 12 (December 2004), 889-903.

[6] Sherwood, K. D.. *Path exploration during code navigation*. The University of British Columbia, 2008.

[7] Bragdon, A., Zeleznik, R., Reiss, S. P. et al. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintanence. In *Proceedings of CHI 2010*.

[8] Bragdon, A., Reiss, S. P., Zeleznik, R. et al. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of ICSE 2010*.

[9] Reiss, Steven P. Simplifying data integration: the design of the Desert software development environment. In *18th International Conference on Software Engineeering* (1996), 398-407.

[10] Reiss, Steven P. The Desert environment. *ToSEM*, 8, 4 (1999), 297-342.

[11] Nackman, Lee R. An overview of Montana. *IBM Research* (1996).

[12] Stockton, R. and Kramer, N. *The Sheets hypercode editor*. CMU, 1993.

[13] Coblenz, M., Ko, A., and Myers, B.. JASPER: an Eclipse plug-in to facilitate software maintenance tasks. In *OOPSLA Workshop on Eclipse Technology* (2006), 65-69.

[14] Kersten, M. and Murphy, G. C. Mylar: a degree-of-interest model for IDEs. In *AOSD '05* (2005), 1590168.

[15] Reiss, S. P. Tracking source locations. In *Proceedings of ICSE'08*, 11-20.