Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments

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. In this paper we describe the design of a prototype IDE user interface for Java based on working sets. A quantitative evaluation shows that developers could expect to view a sizeable number of functions concurrently with relatively few UI operations. A qualitative user evaluation with 23 professional developers indicates a high level of excitement, interest, and potential benefits and uses.

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 simultaneously. This requires the programmer to manually and repeatedly perform numerous interactions to place, resize, scroll, and reflow 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 simultaneously, 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. Bubbles exist in a large virtual space where a cluster of bubbles comprises a concurrently visible working set.

In [4] we developed the bubble metaphor (recapped in Section 5) and associated interaction primitives as part of an iterative design and testing process. A further quantitative evaluation showed that users were able to perform complex code understanding tasks significantly more efficiently when using bubbles than when using Eclipse due to reduced navigation.

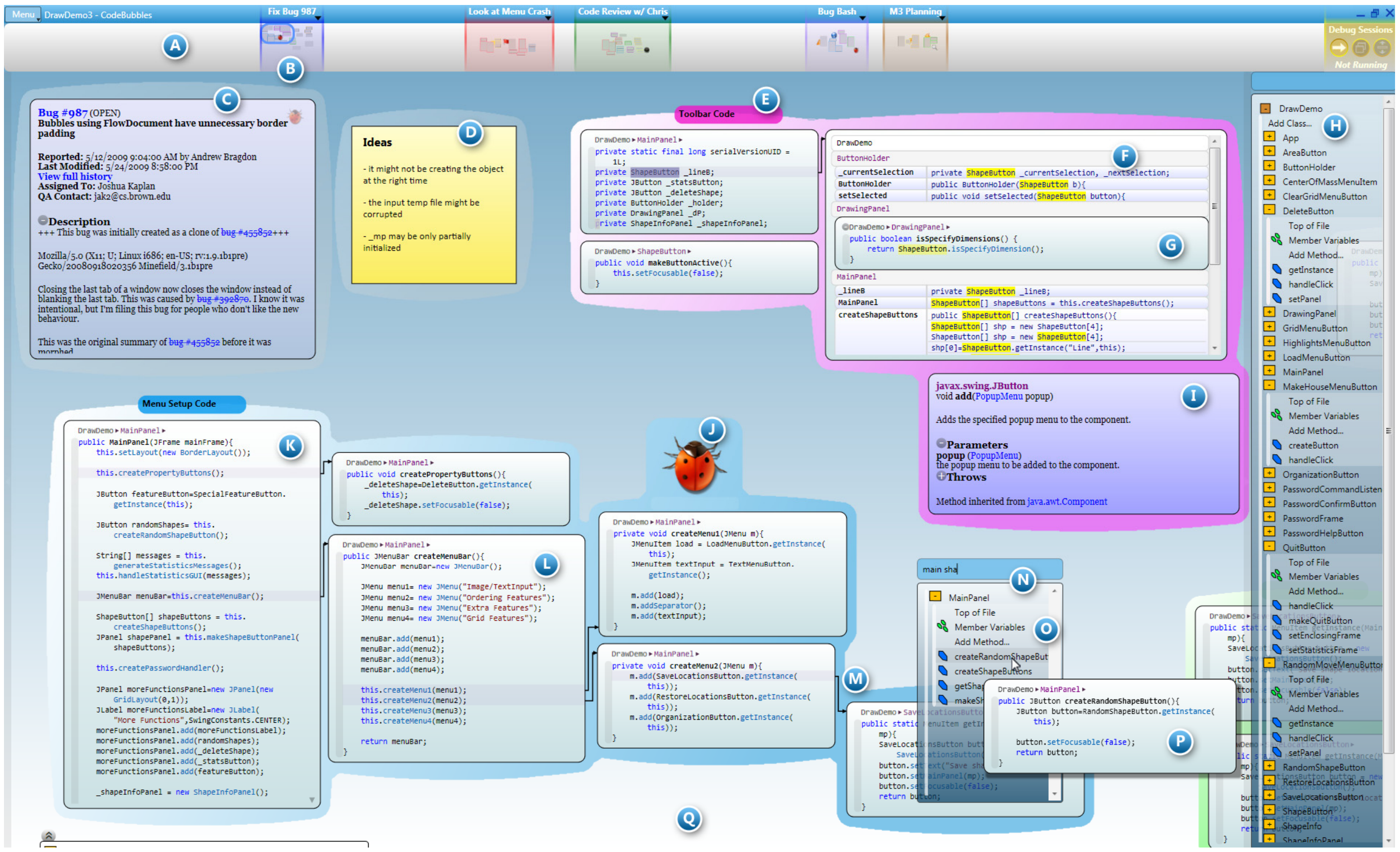
In this paper, we extend the bubble metaphor to build a prototype working set-based user interface for an IDE and constituent tools. The goal of this paper is to explore and understand the merits, usability considerations, and potential uses of this novel user interface paradigm at the IDE system level. Since the problem space addressed by contemporary IDEs is large, we chose a handful of core areas to be: high-impact and broadly applicable to the majority of developers; to represent a sizeable part of the most commonly used functionality; and based on our formative evaluations in developing the metaphor [4] as an initial starting point, including:

- Reading, editing and navigating source code
- Tagging and discovering related functions
- Interruption recovery and multitasking
- Breakpoint debugging (single-threaded)
- Sharing and explaining information

Although this list is by no means exhaustive, we feel this focus is sufficiently broad to shed light on the merit, usability issues, and potential uses of a working set-based IDE user interface.

The contributions of this paper are threefold:

- The design of a novel IDE user interface, and associated development tools, based on working sets of bubbles that has the potential to improve a broad range of development tasks



Bug #987 (OPEN)
Bubbles using FlowDocument have unnecessary border padding

Reported: 5/12/2009 9:04:00 AM by Andrew Bragdon
Last Modified: 5/24/2009 8:58:00 PM
View full history
Assigned To: Joshua Kaplan
QA Contact: jak2@cs.brown.edu

Description
+++ This bug was initially created as a clone of ~~bug #455852~~+++

Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1b1pre) Gecko/20080918020356 Minefield/3.1b1pre

Closing the last tab of a window now closes the window instead of blanking the last tab. This was caused by ~~bug #392870~~. I know it was intentional, but I'm filing this bug for people who don't like the new behaviour.

This was the original summary of ~~bug #455852~~ before it was morphed

Ideas

- it might not be creating the object at the right time

- the input temp file might be corrupted

- _mp may be only partially initialized

```
DrawDemo ▸ MainPanel ▸
private static final long serialVersionUID =
    1L;
private ShapeButton _lineB;
private JButton _statsButton;
private JButton _deleteShape;
private ButtonHolder _holder;
private DrawingPanel _dP;
private ShapeInfoPanel _shapeInfoPanel;
```

```
DrawDemo ▸ ShapeButton ▸
public void makeButtonActive(){
    this.setFocusable(false);
}
```

```
DrawDemo
ButtonHolder
_currentSelection    private ShapeButton _currentSelection, _nextSelection;
ButtonHolder         public ButtonHolder(ShapeButton b){
setSelected          public void setSelected(ShapeButton button){
DrawingPanel
DrawDemo ▸ DrawingPanel ▸
public boolean isSpecifyDimensions() {
    return ShapeButton.isSpecifyDimension();
}
MainPanel
_lineB               private ShapeButton _lineB;
MainPanel            ShapeButton[] shapeButtons = this.createShapeButtons();
createShapeButtons   public ShapeButton[] createShapeButtons(){
                     ShapeButton[] shp = new ShapeButton[4];
                     ShapeButton[] shp = new ShapeButton[4];
                     shp[0]=ShapeButton.getInstance("Line",this);
```

javax.swing.JButton
void **add**(PopupMenu popup)

Adds the specified popup menu to the component.

Parameters
popup (PopupMenu)
the popup menu to be added to the component.
Throws

Method inherited from java.awt.Component

```
DrawDemo ▸ MainPanel ▸
public MainPanel(JFrame mainFrame){
    this.setLayout(new BorderLayout());

    this.createPropertyButtons();

    JButton featureButton=SpecialFeatureButton.
        getInstance(this);

    JButton randomShapes= this.
        createRandomShapeButton();

    String[] messages = this.
        generateStatisticsMessages();
    this.handleStatisticsGUI(messages);

    JMenuBar menuBar=this.createMenuBar();

    ShapeButton[] shapeButtons = this.
        createShapeButtons();
    JPanel shapePanel = this.makeShapeButtonPanel(
        shapeButtons);

    this.createPasswordHandler();

    JPanel moreFunctionsPanel=new JPanel(new
        GridLayout(0,1));
    JLabel moreFunctionsLabel=new JLabel(
        "More Functions",SwingConstants.CENTER);
    moreFunctionsPanel.add(moreFunctionsLabel);
    moreFunctionsPanel.add(randomShapes);
    moreFunctionsPanel.add(_deleteShape);
    moreFunctionsPanel.add(_statsButton);
    moreFunctionsPanel.add(featureButton);

    _shapeInfoPanel = new ShapeInfoPanel();
```

```
DrawDemo ▸ MainPanel ▸
public void createPropertyButtons(){
    _deleteShape=DeleteButton.getInstance(
        this);
    _deleteShape.setFocusable(false);
}
```

```
DrawDemo ▸ MainPanel ▸
public JMenuBar createMenuBar(){
    JMenuBar menuBar=new JMenuBar();

    JMenu menu1= new JMenu("Image/TextInput");
    JMenu menu2= new JMenu("Ordering Features");
    JMenu menu3= new JMenu("Extra Features");
    JMenu menu4= new JMenu("Grid Features");

    menuBar.add(menu1);
    menuBar.add(menu2);
    menuBar.add(menu3);
    menuBar.add(menu4);

    this.createMenu1(menu1);
    this.createMenu2(menu2);
    this.createMenu3(menu3);
    this.createMenu4(menu4);

    return menuBar;
}
```

```
DrawDemo ▸ MainPanel ▸
private void createMenu1(JMenu m){
    JMenuItem load = LoadMenuButton.getInstance(
        this);
    JMenuItem textInput = TextMenuButton.
        getInstance();

    m.add(load);
    m.addSeparator();
    m.add(textInput);
}
```

```
DrawDemo ▸ MainPanel ▸
private void createMenu2(JMenu m){
    m.add(SaveLocationsButton.getInstance(
        this));
    m.add(RestoreLocationsButton.getInstance(
        this));
    m.add(OrganizationButton.getInstance(
        this));
}
```

```
DrawDemo ▸ MainPanel ▸
public JButton createRandomShapeButton(){
    JButton button=RandomShapeButton.getInstance(
        this);

    button.setFocusable(false);
    return button;
}
```

Figure 1. *The Code Bubbles IDE. See section 2 below, Scenario, and section 6 below, IDE User Interface, below. Resolution: 1920x1200 (space reserved for taskbar).*

- A quantitative analysis of the applicability and scalability of Code Bubbles for sizeable open-source projects
- A pilot evaluation with 23 professional developers indicating a very high level of excitement and interest, pointing to potential improvements, and motivating future work

2. SCENARIO

Here we describe a usage scenario of the system as a whole. A narrated video scenario is available online[1]. Screenshots of the Code Bubbles IDE interface are shown in Figures 1 and 2; note that the screenshots are intended to disclose a wide array of features and so do not directly match the scenario narrative.

Jane is planning to change the image uploading feature of a large content management system written in Java to supplement the existing manual image resizing feature with support for automatic resizing. The image uploading feature is complex, spanning dozens of functions across twelve classes. Jane, who is not that familiar with the large code base, wants to avoid being confused by reading code unrelated to the change.

Jane knows that the image upload begins at a specific event handler, so she clicks to bring up the popup search box (Figure 1-N) and types several substrings which filter the list of classes and methods (1-O). She only has to type a few characters to isolate and then open the method she wants as a code bubble (1-K). She right-clicks on method calls within the bubble to open the called functions in separate bubbles (1-L) that group together, but do not overlap; each calling relationship is automatically represented with a connecting arrow (1-M). Within seconds, Jane has opened a working set of 6 method bubbles related to image resizing.

Interested in a particular variable, she executes Find All References on a variable within a bubble to create a bubble stack (1-F) next to the bubble – any overlapped bubbles are automatically repositioned to avoid overlap. The bubble stack shows the lines of code containing the references. Jane can hover over these lines to preview their containing methods or click on them to expand inline (1-G). When she finds the one she wants, she dismisses the rest of the stack while leaving the bubble of interest.

Jane instinctively moves related bubbles near each other to gain an overview of the code structure. The automatic layout manager ensures that any bubbles obscured are automatically moved to eliminate overlap. Moving the bubbles together implicitly creates groups, indicated by an automatically assigned colored halo (1-E) that surrounds the contained bubbles. Jane can enter a name for any group by typing into its empty title box; in this case, she decides to name her group of bubbles “Image sizing and measuring,” both to help her remember the purpose of these bubbles now, and to be able find it in the future, if needed.

Just then, her coworker Jim stops by, interrupting her with a question about a feature she had written months ago within the same content management project. She doesn't want to disrupt the layout of her current working set so she looks for an open area in the workspace bar at the top of the display (1-A), which shows a bird’s eye view of the entire workspace, and clicks to jump to an unused area. She combines popup searching for methods by name with clicking on method calls to quickly create a panorama of the code related to Jim’s question. She zooms in, making it easier for Jim to read the code over her shoulder, and points out the code structure. Jim is confused by one point, so she places a note bubble (1-D) next to the methods and types some items for him to look up later. When they are done, Jane uses the share feature to e-mail Jim the working set of bubbles as an XML attachment that he can later open in his own Code Bubbles instance.

[1] http://www.cs.brown.edu/people/acb/videos/codebubbles.mov

Jane then clicks on her previous working set in the workspace bar (1-B) to return to her previous task. All of the methods and method connections are where she left them; since she had instinctively arranged them to reflect her personal view of her code and task structure, it is relatively easy for her to regain her train of thought and continue working. Jane edits her code directly in the bubbles, with new bubbles budding off for new functions.

To test her changes, she clicks to place breakpoints and presses F11 to start a debug session. When the breakpoint is hit, the workspace pans to an empty region and displays the bubble containing the corresponding code (2-N). Jane steps into a function which automatically opens a second bubble (2-O) to the right of the method that called it with an arrow between the two. As she steps into another method, a third bubble opens (2-P). Jane inspects the contents of a particular object variable by choosing Inspect Data from its context menu, opening a data structure bubble (2-Q) displaying the values of all the fields in the object. Jane wants to focus on only two fields from this bubble which she "tears out" into independent bubbles to view them side-by-side. She realizes that there is a corner case that she hadn't anticipated caused by an unexpected sequence among a set of methods. Jane runs again, this time inputting a different database, and repeats her debugging steps. Since the new debug session (2-A) is visible side-by-side with the previous debug session (2-M), she can compare the values of the data structure from the previous run (2-R) with the current run (2-F). After Jane finds and fixes the problem, she creates a note bubble next to the methods involved in the unusual calling sequence to remind her of this corner case the next time she edits any one of these methods.

3. BACKGROUND AND RELATED WORK

The overall motivation for a new user interface for traditional file-based programming is based on many studies that have shown the difficulties with existing environments [5] [2] [6] [7]. These and other studies have shown that programmers spend a significant amount of time navigating code, that interruptions reduce efficiency, and that programmers interact in terms of working sets that contain the context for their activities. Code Bubbles is an attempt to provide a user interface that facilitates these activities.

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 taken in Desert [8] [9] and it can also be seen in IBM's Visual Age environments [10] and in the Sheets environment from CMU [11]. All these were loosely based on non-file based programming environments such as Xerox's Smalltalk and its successors, including Squeak [12], various versions of Lisp, SELF [13], and visual languages such as NI's LabView. In the Visual Age environment, the system maintained a repository of abstract syntax trees derived from the original program files, and the user edited views built from the tree. Desert took this a step further and allowed the definition of a wide variety of different fragments whose primary representation was their original file, but which could be edited separately. Moreover, Desert supported the notion of virtual files, displays that contain fragments pulled from a variety of files that can be edited and saved, with the edits going back to the original files. These environments did not make use of syntax-aware reflow, automatic layout constraints, fragment groups and annotations, or a continuous virtual workspace which can be subdivided into sections.

Bubbles make use of code elision and code reflow. Most modern environments offer a limited form of code elision [14]. For example, Eclipse will elide multiple import lines into a single line and will let functions and classes be elided manually to their declaration; Eclipse also provides a Formatting command which reflows code to fit. Other environments attempt to do so automatically, providing a fisheye view of the source code [15] focused on the current line of code. Another approach is that of JASPER which attempts to display small read-only views that represent the user's current task as a means for navigation [16].

The problems inherent to navigation in IDEs have been recognized for some time. A number of tools have been developed to add navigation aids to existing file-based environments, for example Hipikat [17], NavTracks [18], Mylar [19] [20], TeamTracks [21], TaskTracer [22], Idewaypoint [23], NaCIN [24], FEAT [25], ConcernMapper [26], and Creole [27]. Navigation in the bubbles paradigm is done by the programmer creating bubble groups and bubble layouts corresponding to working sets, independent of file. We have incorporated some of the techniques from the various navigation tools to assist the programmer in setting up working sets, for example, manually creating bubble groups of related items based on program structure. These navigation tools focus on identifying working sets, whereas we focus on displaying working sets concurrently. Many of the advanced techniques pioneered by these tools, e.g. determining working sets based on navigation, modification histories, or a degree-of-interest model, are potentially complementary to our approach and could be used with the bubbles approach to generate working sets.

The notion of cross-cutting concerns that we deal with using a fragment-based view of the code, is addressed at the language level in aspect oriented programming. While our approach does not currently address multiple concerns within a method, it provides a much lighter-weight mechanism that can be applied to existing code bases.

Finally, we note the techniques used in bubbles apply equally well to debugging while previous work has concentrated on code navigation for editing and understanding. Moreover, debugger extensions, such as WhyLine [28], would have a natural implementation in terms of bubbles, with the various recommendations made by the system being represented as separate bubbles.

4. ARCHITECTURE AND BACKEND

To experiment with the bubbles paradigm, we implemented a prototype for Java suitable for experimentation, user studies, and evaluation. The front end user interface for the prototype is implemented using Microsoft Windows Presentation Foundation. The back-end uses Eclipse by creating a plug-in that accepts requests and provides feedback to the front end using a message bus similar to that of the FIELD environment [29].

The Eclipse plug-in translates many of the Eclipse callbacks into events that are sent to the front end, for example all changes in breakpoints, all execution events, and all resource changes. The plug-in also lets the front end access many Eclipse facilities such as Java search, autocompletion, and refactoring. The plug-in also uses a pattern-based algorithm to provide the front end with the information needed for code elision.

5. THE BUBBLES METAPHOR

The basis for the user interface of our IDE is the bubble metaphor described fully in [4]; in this section we will briefly recap the metaphor and then in the next section present the extensions we make in this paper 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. The bubbles 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
- 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, thus, for example, if the user resizes a bubble, causing reflow to change in that bubble, the underlying file is not affected. The user may also open a function multiple times in separate bubbles; edits in one bubble are automatically propagated to other instances.

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 (see [4] for algorithm description).

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 breadcrumb 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.

In addition to these core display concerns, the bubbles metaphor introduces several other useful facilities. 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).

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.

A separate type of bubble called a bubble stack (1-F) is 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.

6. IDE USER INTERFACE

Building on the bubbles metaphor as a foundation, we have developed a functional IDE user interface, described in this section, 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.

6.1 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.

Finally, within bubbles, programmers can edit code in much the same way they do with a conventional editor. If needed, they can even bring up 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. To do this, users insert a new line at the bottom of a method in the desired class, and begin typing the method's declaration; as the declaration is typed 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-completion, and further augmented it with a new working set-oriented technique. Instead of using auto-completion only to find existing signatures, programmers can type in a new method signature not in the list and create an empty bubble for that new method to be filled in later.

6.2 Building Heterogeneous Working Sets

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 back-end 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 open for each function in the path, and are inserted between the two existing bubbles. In addition, a number is displayed to the right of each bubble, indicating the number of functions called from that bubble that are also part of a path to the end function. Clicking on the number will display a list of these functions; clicking will open that function, which can be further explored.

6.3 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.

6.4 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 [4] that supports the definition of working sets for a particular task or goal. These working sets will typically include several bubble groups and related information.

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.

6.5 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.

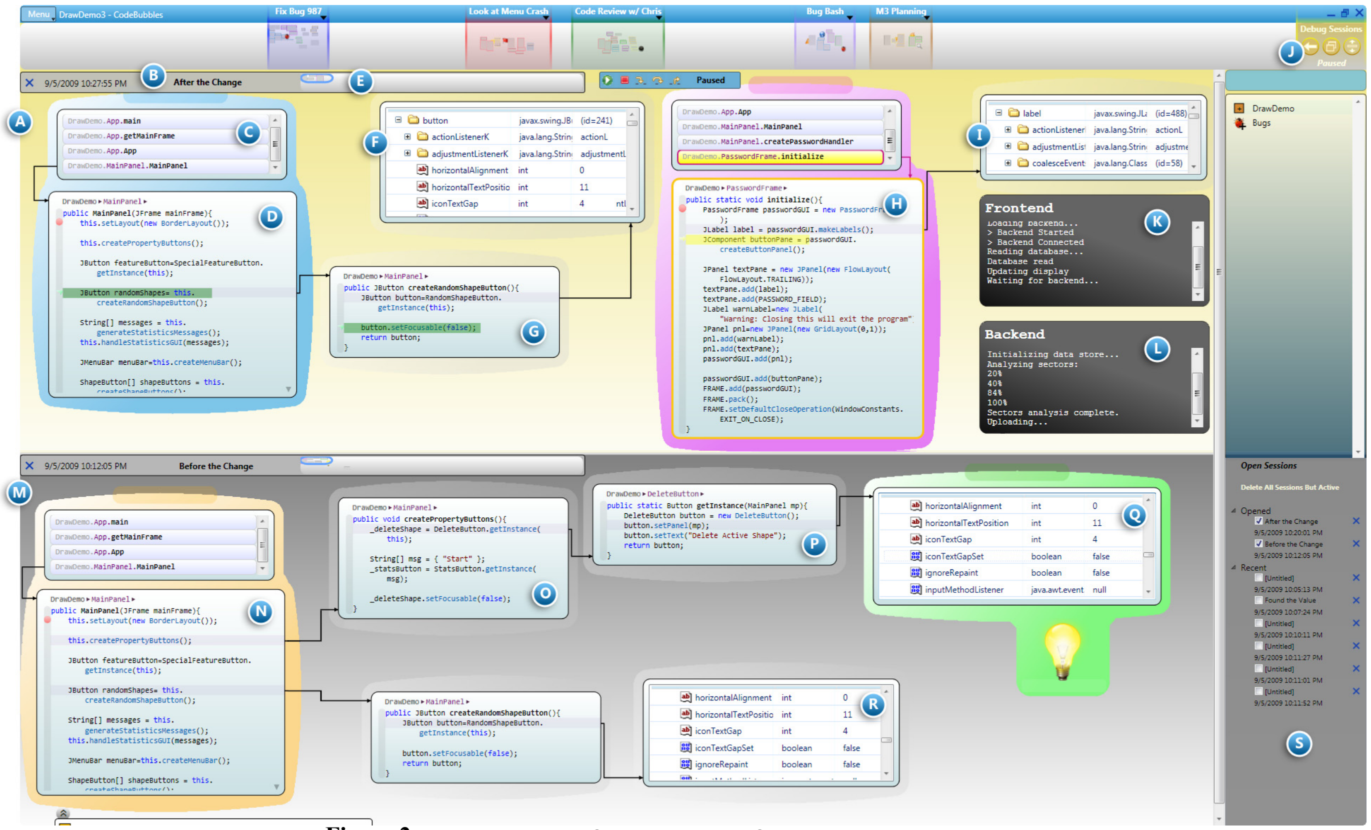
Right clicking on a variable brings up a data structure bubble (2-I) showing the type, name and values of the selected object. These bubbles can be further expanded to show nested values. Typically, these bubbles are updated dynamically as the program executes. However, the user can either freeze a display, 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).

Bubbles are also used to display console output during a debugging session. In addition to a standard console, we support multiple virtual consoles (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.

6.6 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 (stored as XML). 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



```
DrawDemo.App.main
DrawDemo.App.getMainFrame
DrawDemo.App.App
DrawDemo.MainPanel.MainPanel
```

```
DrawDemo▸MainPanel▸
public MainPanel(JFrame mainFrame){
    this.setLayout(new BorderLayout());

    this.createPropertyButtons();

    JButton featureButton=SpecialFeatureButton.
        getInstance(this);

    JButton randomShapes= this.
        createRandomShapeButton();

    String[] messages = this.
        generateStatisticsMessages();
    this.handleStatisticsGUI(messages);

    JMenuBar menuBar=this.createMenuBar();

    ShapeButton[] shapeButtons = this.
```

```
⊟ button                   javax.swing.JB  (id=241)
  ⊞ actionListenerK        java.lang.Strin actionL
  ⊞ adjustmentListenerK    java.lang.Strin adjustmentL
    horizontalAlignment    int             0
    horizontalTextPositio  int             11
    iconTextGap            int             4
```

```
DrawDemo▸MainPanel▸
public JButton createRandomShapeButton(){
    JButton button=RandomShapeButton.
        getInstance(this);

    button.setFocusable(false);
    return button;
}
```

```
DrawDemo.App.App
DrawDemo.MainPanel.MainPanel
DrawDemo.MainPanel.createPasswordHandler
DrawDemo.PasswordFrame.initialize
```

```
DrawDemo▸PasswordFrame▸
public static void initialize(){
    PasswordFrame passwordGUI = new PasswordFr
        );
    JLabel label = passwordGUI.makeLabels();
    JComponent buttonPane = passwordGUI.
        createButtonPanel();

    JPanel textPane = new JPanel(new FlowLayout(
        FlowLayout.TRAILING));
    textPane.add(label);
    textPane.add(PASSWORD_FIELD);
    JLabel warnLabel=new JLabel(
        "Warning: Closing this will exit the program"
    JPanel pnl=new JPanel(new GridLayout(0,1));
    pnl.add(warnLabel);
    pnl.add(textPane);
    passwordGUI.add(pnl);

    passwordGUI.add(buttonPane);
    FRAME.add(passwordGUI);
    FRAME.pack();
    FRAME.setDefaultCloseOperation(WindowConstants.
        EXIT_ON_CLOSE);
}
```

```
⊟ label               javax.swing.JL  (id=488)
  ⊞ actionListener    java.lang.Strin actionL
  ⊞ adjustmentList    java.lang.Strin adjustme
  ⊞ coalesceEvent     java.lang.Class (id=58)
```

```
Frontend
Loading backend...
> Backend Started
> Backend Connected
Reading database...
Database read
Updating display
Waiting for backend...
```

```
Backend
Initializing data store...
Analyzing sectors:
20%
40%
84%
100%
Sectors analysis complete.
Uploading...
```

```
DrawDemo▸MainPanel▸
public void createPropertyButtons(){
    _deleteShape = DeleteButton.getInstance(
        this);

    String[] msg = { "Start" };
    _statsButton = StatsButton.getInstance(
        msg);

    _deleteShape.setFocusable(false);
}
```

```
DrawDemo▸DeleteButton▸
public static Button getInstance(MainPanel mp){
    DeleteButton button = new DeleteButton();
    button.setPanel(mp);
    button.setText("Delete Active Shape");
    return button;
}
```

```
horizontalAlignment    int              0
horizontalTextPosition int              11
iconTextGap            int              4
iconTextGapSet         boolean          false
ignoreRepaint          boolean          false
inputMethodListener    java.awt.event   null
```

```
DrawDemo▸MainPanel▸
public JButton createRandomShapeButton(){
    JButton button=RandomShapeButton.
        getInstance(this);

    button.setFocusable(false);
    return button;
}
```

```
horizontalAlignment    int      0
horizontalTextPositio  int      11
iconTextGap            int      4
iconTextGapSet         boolean  false
ignoreRepaint          boolean  false
```

Figure 2. *Debugging with the Code Bubbles IDE. See section 6.5, Debugging, above.*

having them reload the bubble configuration in their own workspace.

7. EVALUATION

In [4] we showed that the bubbles metaphor could improve code understanding performance. In this paper, our focus is on how the metaphor scales and its utility for real systems. To this end, we had two principal questions:

- How well does Code Bubbles perform on real applications in terms of simultaneous readability?
- How do experienced professional developers evaluate the different features we have developed as they perform representative tasks in a controlled environment?

7.1 Quantitative Analysis

Our first study compared the Code Bubbles IDE with an existing IDE (Eclipse 3.4.2) in terms of scalability. We conducted a quantitative performance analysis of the following questions:

- How many functions can one see on screen simultaneously in Code Bubbles vs. Eclipse?
- How many UI operations must one use to create concurrently-visible working sets in Code Bubbles versus Eclipse?

7.1.1 Methodology and Metrics

For this evaluation we looked at three large Java-based open source applications, ArgoUML, a UML designer (155,603 lines of code [LOC]), jEdit, a text editor (109,925 LOC), and jForum, a forum web application (30,064 LOC). We used four conditions, Code Bubbles with and without vertical elision (CB, CB+VE), Eclipse with and without vertical elision (E, E+VE).

For each application, we considered the following test cases: ***Worst case (WC):*** the longest (largest LOC) functions in the application. ***Random case (RC):*** a random selection of application functions. ***Typical case (TC):*** all of the functions involved in a specific feature from the application, chosen to be representative.

We compute three metrics for each of these cases: ***Simultaneously visible functions (SVF)***: the number of function declarations one can see on screen simultaneously, without scrolling horizontally or vertically. ***UI operations (UIOp):*** minimum number of UI operations required to create a task relevant display with SVF functions, where a UI operation is a command involved, e.g. resize, move, scroll, "New Editor," etc. ***Normalized UI operations (NUIOp):*** UIOp divided by SVF.

A systematic approach was taken to analyze the mentioned metrics. First, both of the editors were standardized, using the same font (Consolas 8), only package explorer pane opened (others closed), and a 1900x1200 24" display. For both systems, we simulated an optimal user who used an optimal UI manipulation strategy to open as many functions as could fit on one screen without overlap, and minimal scrolling. In Eclipse, tabs were docked into panes as needed. Code Bubbles automatically reflows text if the length of a line exceeds the width of the bubble. In Eclipse we manually invoked its code formatting feature to fit the code without horizontal scrolling. In Eclipse, the "New Editor" command was used as needed to open files multiple times. When no more functions from the test case could fit (opened in same order for both conditions), we stopped the trial and computed UIOp and SVF.

7.1.2 Results and Analysis

Results are presented in Table 1. Code Bubbles was able to show more functions simultaneously (SVF) in every case. In the worst case, the average increase from Eclipse to Code Bubbles (cols. 1, 2) was 83.33%; in the random case, 49.04%; in the typical case, 49.53%. With vertical elision (cols. 3, 4), in the worst case, the average increase was 36.11%, in the random case, 28.55%; in the typical case, 54.54%. This increase was likely due to the reduced chrome in Code Bubbles, and differences in the reflow algorithm. In real terms, developers would be able to see more methods in Code Bubbles (col. 2), a sizeable number of functions, 11-17, in the random/typical cases; the worst case provides a lower-bound of 3-4 methods.

		SVF *(higher is better)*				***NUIOp*** *(lower is better)*			
		E ①	CB ②	E+VE ③	CB+VE ④	E ⑤	CB ⑥	E+VE ⑦	CB+VE ⑧
ArgoUML	WC	2	**3**	3	**4**	**1.50**	2.00	**1.67**	1.75
	RC	10	**16**	16	**21**	2.60	**1.25**	2.81	**1.33**
	TC	9	**14**	14	**22**	2.56	**1.29**	2.81	**1.23**
jEdit	WC	2	**4**	4	**6**	2.00	2.00	2.20	**1.67**
	RC	11	**16**	16	**20**	3.00	**1.19**	2.71	**1.25**
	TC	9	**14**	14	**20**	2.56	**1.29**	2.88	**1.20**
jForum	WC	2	**4**	4	**5**	**1.50**	2.00	3.50	**2.00**
	RC	12	**17**	17	**22**	2.67	**1.29**	2.88	**1.36**
	TC	8	**11**	11	**18**	2.50	**1.45**	2.73	**1.28**

Table 1. *Results of quantitative analysis of Eclipse (E) and Code Bubbles (CB) with and without vertical elision (VE); column numbers circled.*

Vertical elision increased SVF for both conditions, suggesting it may be a beneficial alternative to scrolling vertically; in Eclipse (cols. 1, 3) the increase ranged from 37.50% to 100.00%; in Code Bubbles (cols. 2, 4), the increase ranged from 25.0% to 63.63%. However, it appears that vertical elision could be an optional feature for users concerned about hiding code, since SVF was still sizeable without it.

Perhaps the most important impact of Code Bubbles, however, was in the area of UI operations (cols. 5, 6). In the random case Code Bubbles reduced normalized UI operations by an average of 54.64% from Eclipse; in the typical case 47.07%. Interestingly, when looking at specific worst case trials, Eclipse outperformed Code Bubbles in 3 out of 6 trials; however the average difference in the 6 worst case trials was less than 1%. This was likely due to the small size of the working set in these cases, creating noise (2-4 functions for Eclipse).

These results indicate that with real-world code, one can expect to be able to see a sizeable number of functions side-by-side in Code Bubbles (cols. 2, 4), while expending less than half the number of UI operations to create such a view than required by Eclipse (close to 1 UI operation/function in most cases).

7.2 Qualitative User Evaluation

In addition to a controlled quantitative evaluation, we wanted to get detailed, qualitative feedback from professional developers. Professional developers are a very demanding customer. They are expert users with significant experience using existing IDE(s). In addition, they often pride themselves in working efficiently. Therefore, one might reasonably expect them to be highly critical of any new and fundamentally different application or user interface, and thus an ideal population for a qualitative study.

In the study we asked the developers to perform a series of tasks using Code Bubbles and to give detailed critical feedback based on this hands-on experience. The feedback's purpose was to improve the design, to gauge its value to developers, and to understand how developers might use the system in practice. The system is still a prototype and is not ready for a production setting; therefore we chose to do a more controlled lab study.

As this was a lab study, there are obvious limitations on the study's generality; however we believe it was sufficiently representative for the experienced developers who participated to generalize to their workplace. We previously conducted an initial qualitative study of the core Code Bubbles metaphor [4]; in this study we examine the additional features contributed in this paper.

7.2.1 Participants and Methodology

We recruited 23 professional developers (compensated) from the Providence, RI and Boston, MA area (21 male, 2 female; mean age 33.04, S.D. 9.47; reporting an average of approximately 10 years of industry experience). We advertised the study with Internet ads so as to gather a diverse sampling of the developer population; they hailed from organizations ranging from large enterprises to small firms, and from a range of industries. Participants also used a range of tools, including Eclipse, NetBeans, IntelliJ IDEA, Visual Studio, VIM, XEmacs, and Notepad++. We required that participants use the Java programming language on a regular basis, and to be actively employed in a software development role; no members of our research group participated in the study.

After filling out a pre-questionnaire, participants were read an introductory statement, introducing them to the user study, and asking them to think aloud throughout the tasks they would be working on. Participants were also asked to draw from past development experiences in communicating their ideas and thoughts. Participants were then asked to perform a series of tasks (see below) using Code Bubbles. Participants were instructed on how to use relevant features during each task as needed. The study lasted approximately 1.5 hours. All trials used a 24" monitor running at 1920x1200x32-bit; total computer + monitor cost < $1,000 (US).

7.2.2 Task Context and Tasks

We evaluated the performance of Code Bubbles using a vector-based drawing application we created, similar at a high level to that used in [2] in Java, called ShapeDraw. The system is comprised of 44 classes, 280 methods, and 2,658 lines of code (mean of 6.36 lines/function, S.D. of 13.20). To control for *a priori* knowledge of API libraries such as Java Swing, we wrapped all non-trivial APIs so as to not directly expose the programmer to such APIs (we did not wrap common data structures, such as LinkedList); we also structured the code to not involve algorithms, protocols, databases, or file formats. Such knowledge is inherently involved in working with open source applications which use a variety of libraries, technologies, algorithms, etc. which some participants might be less experienced with than others.

Participants worked on six tasks, each task requiring participants to use different features of the system:

Code comparison**:** participants were asked to compare 5 methods to understand the differences and relationships between them.

Code understanding**:** participants were given a particular method as a starting point, and asked to understand the feature located there by navigating and reading through part of the call graph originating there (6 methods), and by examining an instance variable's references (3 methods). Participants were then asked to add additional code to an existing method, and create a new method.

Simulated interruption and interruption recovery: participants were asked to stop the code understanding task while only part way complete, and switch to work on a different code understanding task involving 3 methods. Once this high priority task was complete, participants were asked to resume the interrupted task.

Debugging and dynamic program understanding: participants were asked to investigate several bugs, and identify the source of the problem: a runtime exception, whether two data structures contain similar values to each other at different points in the program's execution, and whether a simple numerical calculation is outputting the correct values.

Sharing information: participants were tasked with sharing the results and conclusions from their debugging session with another person who was not on-site.

Debugging session comparison: participants were tasked with repeating the data structure subtask, but this time inputting slightly different interactions into the test program.

7.3 Observations

On the whole, developers were uniformly very positive, rating the system 4.33 ± 0.26 (95% confidence interval) on a 5-point Likert scale, in which 5.0 was "very convenient." Of all the participants, only one (a software architect who spent less than 10% of his time working with code) said they would not consider adopting the system as their primary IDE. When it came to learning to use the system, developers rated it on average 3.39 ± 0.38, in which 3.0 was "neither easier nor harder" than other software.

Multiple developers asked whether specific build configurations and plug-ins would be compatible; they were pleased to note that since Code Bubbles is built on Eclipse, Eclipse is running in the background and can handle many such tasks. They also noted the lack of XML file support, citing this as an important needed feature.

Developers were very positive about reading and editing code in bubbles. They felt being able to see multiple functions side-by-side was very convenient; rating it an average of 4.7 ± 0.44 where 5.0 was "very convenient." They rated Code Bubbles an average of 4.04 ± 0.36 on a 5-point Likert scale where 5.0 was "much easier," for reviewing code than their preferred IDE. Interestingly, 4 of the 23 developers rated it below a 4.0; three of these participants worked primarily in a software architect role, and mentioned that they did not write significant amounts of code on a day-to-day basis. Developers preferred bubbles to Eclipse's tabs and found it useful for exploring new methods and classes.

7.3.1 Files and Writing New Code

When it came to writing new code in Code Bubbles, developers felt they could use bubbles for most tasks, but there might still be occasional tasks in which they would want to bring up a class file. Many developers further mentioned that in the code bases in use at their firm, classes tended to be very large and that seeing the class as a file was not particularly useful most of the time. Three developers reported that in Eclipse they tend to use the package explorer for browsing rather than browsing in the file view, and so their current workflow would not be impacted significantly moving to bubbles. Although they liked the techniques we developed for writing new code, including budding and adding methods with auto-complete, some developers still felt there might be situations in which they would want to edit a whole file, most commonly for writing a new class from scratch, but also for skimming through and "tearing out" relevant methods into bubbles. Developers also liked the ability to see an entire class through one menu item click. These requests, and the fact that developers did not seem to view function and class bubbles as mutually exclusive, led us to develop the class bubble to handle situations in which seeing a whole class was useful early in the development process.

Developers were excited that Code Bubbles encourages short functions. They noted that file-based IDEs can discourage writing short methods because they know that it will create navigation overhead in the future. Developers all perceived shorter methods as a positive thing, with several participants complaining at length about the "mammoth" methods created by their coworkers.

All but one of the developers liked vertical elision, commenting that it made long functions easier to "scan" and read, and further commenting that in long methods they were frequently only interested in a particular section. Reflow, on the other hand, appeared to involve personal preferences. A significant majority of developers liked or did not mind reflow, with some even commenting that it made the code easier to read. However, three developers felt that too much reflow could be distracting and the ability to set a default minimum width for a bubble was needed.

7.3.2 Creating and Grouping Working Sets

Developers were unanimous in their enthusiasm for a working set-based IDE. They liked being able to open different kinds of fragments together in a working set saying, "opening Javadocs is pretty cool. This is beautiful." and "That's cool. I like it. For example, I can use the notes feature to leave myself notes like 'there is a bug in the new drawPanel function'." Developers also liked bug bubbles, one developer saying "[I] could see how [a bug bubble] could become the anchor point of the relevant task." Several developers felt that sometimes bubbles would only be needed for a short time and asked for the ability to open a code or Javadoc bubble in a temporary preview.

Developers were similarly enthusiastic about grouping related bubbles. They thought they would create groups "on the spur of the moment" and had many spontaneous ideas on how they might use groups, including organizing relevant subsets of a large existing feature; comparing two versions of a feature; "color coding who is doing what"; as a supplement to the organization provided by packages and classes; and using them as an onscreen visual association and reminder. They also liked the ability to save and recall groups. In particular, developers liked the idea of using group information to discover related code – either in revisiting their own, or that of others. Developers liked the ability to search for groups by name and contents; several developers went further to suggest the idea of being able to search for groups by the text contents of note bubbles included in a group. One developer suggested adopting a practice in which groups were used to document by example how to do something, e.g. if using a particular component was counter-intuitive, a reusable example could be saved as a group, along with a note bubble explaining the process. One developer thought that it was too easy to make groups; that they should be explicitly defined.

7.3.3 Interruptions and Multi-tasking

Developers liked using the workspace bar to take advantage of a large virtual workspace. They liked the continuous nature of the workspace, complaining that they found discrete virtual desktop managers to be "confusing" and that individual desktops tended to "fill up." Developers appreciated the ability to label areas of the bar, being able to switch between different tasks, and the ease with which they could begin a new task and resume a pre-existing task. They felt working sets could replace their hand-written notes and would help with this recovery process because the set of bubbles combined with notes and flags represented much of what they had to remember for the original task.

Four developers asked for the ability to save and close workspace bar tasks so that they could be used later, further telling us that once they had created a task it would be very useful to return to it later. This led to our implementation of the task shelf. Five developers asked for the ability to open multiple projects in different regions on the workspace bar so that they could easily switch between them and potentially reuse functionality across projects.

7.3.4 Debugging

Developers were excited by the prospect of debugging with bubbles; in some cases even urging us to implement a breakpoint debugger using bubbles early in the experiment before it was shown to them. They rated the debugger highly; an average of 4.55 ± 0.47 where 5.0 is "Very Convenient." All of the developers liked seeing the broader context of multiple items in the stack frame side-by-side, and seeing new bubbles open as step into and breakpoint events caused the call stack to change. Developers did not miss flipping back and forth through the call stack. They further liked being able to open data structure values in bubbles as part of their working set, with many developers commenting that it was useful to be able to see values from more than one location in the call stack simultaneously, allowing them to "compare and contrast." Several developers asked for the ability to collapse parts of the call graph hierarchically using contract/expand widgets, as they were concerned that inactive branches could be irrelevant in some cases and they would want to close them. One developer was initially opposed to the idea of opening a bubble multiple times in a recursive calling sequence, but later changed his mind in favor of seeing the function at "different points in time."

All but one of the developers were enthusiastic about the idea of directing printouts to multiple named console bubbles, mentioning that they liked being able to separate event streams when needed so they could be examined separately, and also mentioning that as the number of printout sources accumulated in a project, it could become difficult to see the printouts they were interested in. One developer said that he made extensive use of logging frameworks, and although he liked seeing multiple consoles side-by-side, he found our implementation limiting.

Developers were also positive about saving debug sessions, mentioning in particular that it was useful to be able to compare information across separate runs of the program, and to revisit debugging information. They perceived value in being able to compare before and after runs of a program. They also mentioned that they sometimes needed to repeat a complete debugging session due to missing or not noting something. Developers also liked the fact that all of the console bubbles are preserved in a saved debug session. Several developers mentioned that since they were now able to see several debug sessions simultaneously they would like the ability to "diff" them, to identify differences in probed data structure values. Developers liked the channels interface we developed for debug sessions and several developers suggested that the same interface could be used to debug multiple threads, or multiple distributed applications, simultaneously.

7.3.5 Sharing and Working Together

Using bubbles to share information was of nearly universal interest to the developers, with nearly all developers spontaneously voicing ideas on different ways this would be useful. Developers liked the fact that the visual representation of bubbles closely mimicked what would be needed for a visual explanation to another person. Five developers suggested using working sets for code reviews. Developers further liked the idea of adding notes or flags to a debug session and then posting it as a PDF to a bug report in a bug tracker. They appreciated the ability of using a working set for explaining code and for transitioning a feature from one person to another, as well as the idea of using working sets to ask questions, get feedback, and make in-person presentations. Developers liked both the ability to send a working set as an XML file, and also as a platform-independent PDF; this led us to add support for PDF export and attaching a working set to a new email with one command.

Five developers also spontaneously suggested using the workspace bar as a way of enabling real-time collaboration and coordination. Developers suggested allocating a section of their workspace to be shared live with someone else, to allow one developer to potentially help another, either for a short or prolonged task, in real-time. Several developers also suggested integrating instant messaging so that a developer could share their workspace and ask a quick question, and get feedback in real-time.

8. LIMITATIONS

While the above studies show that the Code Bubbles approach has promise, there are limitations both with the prototype implementation and in extending the concepts to large-scale development.

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. Developers expressed concern about the effectiveness of bubbles on small monitors. Many features one might expect in a complete IDE are missing: support for programming languages other than Java, portability, GUI designers, unit testing, XML files, HTML designers, database designers, and performance monitoring. The editor provided by Code Bubbles is not as sophisticated as modern program editors such as Eclipse's or Visual Studio's. Also many of the features of Eclipse, for example quick fixes for errors and refactoring other than naming, are not yet implemented.

Storing bubble references in general and working sets in particular is currently done using fully-qualified method signatures. This would be problematic in a real development environment where functions are renamed, deleted or moved among classes. This problem could be ameliorated by storing workspace information using file offsets, and applying the techniques developed in [30].

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. More investigation is needed as to how to assist programmers in developing appropriate debugging working sets for more general debugging problems.

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.

9. DISCUSSION AND FUTURE WORK

On the whole, the high level of interest, excitement and range of usage ideas from developers was a surprising result, given the limitations of our prototype, the radical change from what developers are used to, and the level of experience of the participants. We believe this indicates that developers perceive significant value in a working set-based user interface paradigm for IDEs.

The use of a working set-based user interface paradigm in Code Bubbles appears to have changed the cost structure of using working sets to aid in completing tasks; developers did not have to explicitly create a working set from scratch to use one, rather they "get it for free" as part of their normal workflow. As a result, annotation tools such as groups, flags, notes, connections, etc. are always available, making them something developers can count on regardless of task. This change in cost structure means that working sets can be employed more often, which impacts a variety of tasks, as we will discuss here.

Developers liked being able to offload information from their limited working memory by storing information in the working set in the form of bubbles and annotations. Furthermore, they liked being able to position bubbles for convenience and ease of access, to take advantage of efficiency gains derived from spatial proximity and spatial memory. Developers also liked being able to access their code in a non-architectural, task-oriented structure.

Working sets also facilitated comparison tasks, by allowing developers to compare information in one or more bubbles side-by-side. We observed developers take advantage of this in a range of instances from examining debugging values over time, to understanding a call graph encompassing a handful of functions.

Users felt that annotated working sets can also readily be used to share information with other users. This same value in communicating information to others also applies when a user needs to revisit information; developers liked how working sets support interruption recovery and multi-tasking. Also as a result, persisting working sets also had value for developers, making it easy to revisit and leverage working sets users created in the past.

From our observations we believe that the working set paradigm manifested itself in the wide variety of ingenious and unexpected ideas users voiced on how they might use Code Bubbles. This enthusiasm seemed to reflect the perception that the system could be used to aid in a variety of open-ended tasks, such as code reviews, storing examples as groups, posting PDF working sets as bug reports, transitioning a feature from to another person, etc. We were surprised by the extent to which developers sought to employ the system in solving problems that we had not explicitly designed it to solve, suggesting that if used in the workplace, the system could be adaptable in serving a wide variety of needs.

These results provide a rich basis for future work and research. We are currently investigating different approaches to using bubbles for debugging and means for using bubbles for code sharing and cooperative development. Future directions we are considering include extending bubbles to other aspects of software development, in particular UML-based design, performance analysis, testing; adapting the framework for other languages; and extending the debugging capabilities to provide a more effective environment. In addition, further empirical studies with professional developers and with students are warranted, once sufficient features and stability have been implemented.

10. CONCLUSION

We have presented a novel user interface paradigm for an integrated development environment. This paradigm is based on working sets of fine-grained, editable fragments, or bubbles. We also presented two evaluations of this system to better understand how it might perform, be used, and benefit professional developers. The results indicate that a working set-based IDE, such as Code Bubbles, has the potential to benefit a wide range of development tasks in substantial code bases, including reading, editing and navigating source code, tagging and discovering related functions, interruption recovery and multitasking, breakpoint debugging and sharing and explaining information.

11. 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 IIS-0812382 and CCR-0613162.

12. REFERENCES

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

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

[3] Plumlee, M. D., Ware C. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. *ACM ToCHI*, 13,2 (6/06), 179-209.

[4] Bragdon, A. et al. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintanence. *In Proceedings of CHI 2010*.

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

[6] Robillard, M. P., et al. How effective developers investiage source code: An exploratory study. *IEEE TSE*, 30, 12 (Dec. 2004), 889-903.

[7] Sherwood, K. D. *Path exploration during code navigation*. UBC, 2008.

[8] Reiss, S. P. Simplifying data integration: the design of the Desert software development environment. In *ICSE'96*, 398-407.

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

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

[11] Stockton, R. et al. *The Sheets hypercode editor*. CMU, 1993.

[12] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of OOPSLA'97*, 318-326.

[13] Ungar, D. and Smith, R. B. Self: The power of simplicity. In *Proceedings of OOPSLA'87*, 227-242.

[14] Cockburn, A. et al. Hidden messages: evaluating the efficiency of code elision in program navigation. *Interacting with Computers* (2003), 387-407.

[15] Jakobsen, M. and Hornbaek, K. Evaluating a fisheye view of source code. In *Proceedings of CHI'06*, 377-386.

[16] Coblenz, M. et al. JASPER: An Eclipse plug-in to facilitate software maintenance tasks. In *OOPSLA Workshop on Eclipse Tech. 2006,* 65-69.

[17] Cubranic, D., Murphy G. C. Hipikat: recommending pertinant software development artifacts. In *ICSE'03*, 408-418.

[18] Singer, J., Elves, R., and Storey, M. A. Navtracks: supporting navigation in software. *ICPC'05*, 173-175.

[19] Kersten, M. et al. Mylar: degree-of-interest model for IDEs. *AOSD '05*.

[20] Kersten, M. and Murphy, G. C. Using task context to improve programmer productivity. In *SIGSOFT 06/FSE 14 (2006), 1-11.*

[21] DeLine, R., Czerwinski, M., and Robertson, G. Easing program comprehension by sharing navigation data. *VL/HCC 2005, 241-248.*

[22] Dragunov, A. et al. TaskTracer: a desktop environment to support multi-tasking knowledge workers. *IUI, 2005, 75-82.*

[23] Zhang, J. Idewaypoint: support task-oriented IDE navigation. Univerisity of Victoria, 2006.

[24] Majid, I. et al. NaCIN: an Eclipse plug-in for program navigation-based concern inference. In *OOPSLA Workshop on Eclipse Tech. '05*, 70-74.

[25] Robillard, M. P. Murphy G. C. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *ICSE* '03, 822-823.

[26] Robillard, M. P. et al. ConcernMapper: simple view-based separation of scattered concerns. In *OOPSLA workshop on Eclipse Tech.* (2005), 65-69.

[27] Lintern, R. et al. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In *SoftVIS '03* (2003), 47-56.

[28] Ko, A. J. et al. Debugging Reinvented: Asking and answering why and why not questions about program behavior. In *ICSE'08*, 301-310.

[29] Reiss, S. P. Connection tools using message passing in the FIELD environment. *IEEE Software*, 7, 4 (July 1990), 57-67.

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