Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance

Andrew Bragdon[1], Robert Zeleznik[1], Steven P. Reiss[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, bcz, spr, suman, wcheung, jak2, cjc3, fadeputr}@cs.brown.edu

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

ABSTRACT

Developers spend significant time reading and navigating code fragments spread across multiple locations. The file-based nature of contemporary IDEs makes it prohibitively difficult to create and maintain a simultaneous view of such fragments. We propose a novel user interface metaphor for code understanding based on collections of lightweight, editable fragments called bubbles, which form concurrently visible working sets. We present the results of a qualitative usability evaluation, and the results of a quantitative study which indicates Code Bubbles significantly improved code understanding time, while reducing navigation interactions over a widely-used IDE, for two controlled tasks.

Author Keywords

Multi-view, simultaneous views, source code, bubbles, Java

ACM Classification Keywords

H5.2 Information Interfaces and Presentation: Windowing Systems, Evaluation/Methodology

INTRODUCTION

Studies indicate that programmers spend a significant amount of time reading and navigating code; one study puts the total at 60-90% [1]. Programmers form working sets of one or more code fragments corresponding to places of interest [2]; with larger code bases, these fragments are scattered across multiple methods in multiple classes.

Allowing developers to see, interact with and edit multiple fragments concurrently has the potential to make code understanding and maintenance easier by offloading limited working memory resources and enabling new behaviors. Indeed, [3] has 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. Developers could form working sets to inspect and compare functions to identify commonalities, parallels, and differences; form and inspect working sets to answer specific questions; and navigate unfamiliar code with less fear of "getting lost," since they could glance to be reminded of where they had navigated from.

Because modern integrated development environments (IDEs) are file-based, creating and maintaining views of multiple simultaneously visible fragments is difficult. Programmers must manually and repeatedly perform numerous interactions to place, resize, scroll, and reflow a different file pane/window for each fragment. IDEs are instead optimized for switching among views using tabs, forward/back buttons, etc. Perhaps as a result, programmers may spend an average 35% of their time just navigating among code fragments [2], since they can only see one or two at a time.

We therefore argue for a novel user interface metaphor for reading and editing code, one which is based around creating task-relevant collections of code fragments, allowing the user to see and work with complete working sets.

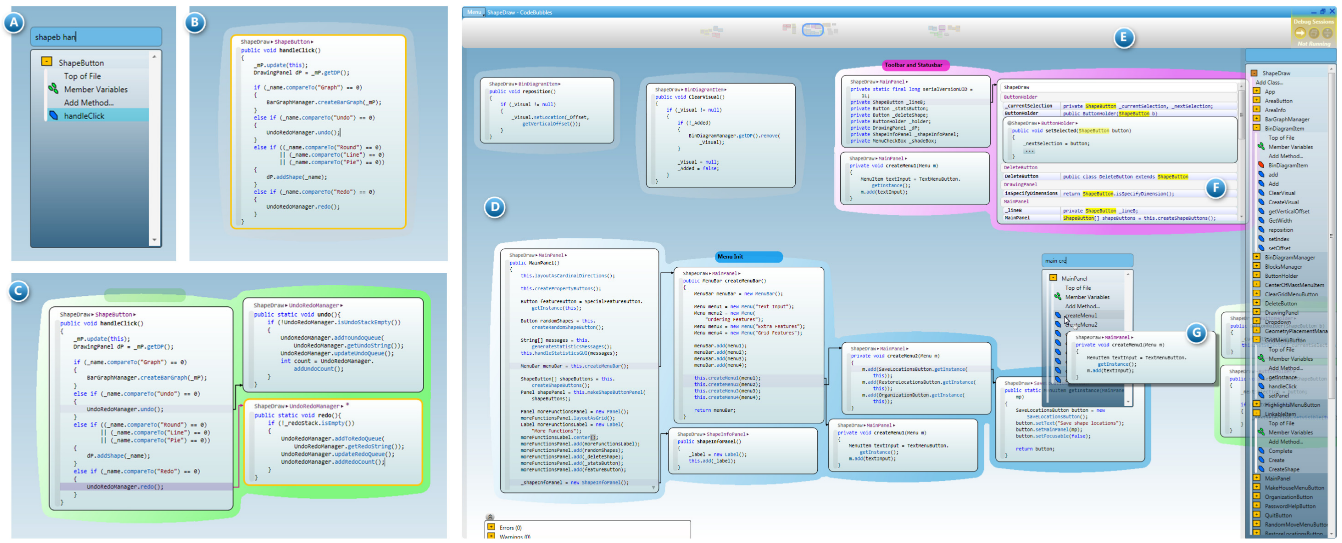
Our approach is founded on the metaphor of a *bubble* – a fully editable and interactive view of a fragment such as a method or collection of member variables. Bubbles, in contrast to windows, have minimal border decoration, avoid clipping their contents by using automatic code reflow and elision, and do not overlap but instead push each other out of the way. Bubbles exist in a large, pannable 2-D virtual space where a cluster of bubbles comprises a concurrently visible working set. See Fig. 1 for a usage scenario.

The contributions of this paper are three-fold: a novel design for a function-based editing interface, Code Bubbles; the results of a qualitative evaluation and accompanying usability discussion of the system; and the results of a quantitative experiment which indicates Code Bubbles significantly reduces the time required to understand code and the number of navigation interactions for two controlled tasks.

BACKGROUND AND RELATED WORK

User interfaces for programming have a long history. The notion of working with program fragments – individual functions, or similar units – was explored by Desert [4] and can be found in IBM's Visual Age [5] and CMU's Sheets [6]. These systems were loosely based on non-file based languages (e.g., Xerox's Smalltalk, Lisp), but none provided either a tiling assistant to avoid fragment overlap or a continuous desktop. These omissions, coupled with a range of UI choices such as not providing automatic code reflow, we believe limited their usability and effectiveness.

Several studies show a range of difficulties with common programming tasks which can be traced to UI designs that complicate access to working sets [7] [2] [8] [9]. Such stu-



```
ShapeDraw▸ShapeButton▸
public void handleClick()
{
    _mP.update(this);
    DrawingPanel dP = _mP.getDP();

    if (_name.compareTo("Graph") == 0)
    {
        BarGraphManager.createBarGraph(_mP);
    }
    else if (_name.compareTo("Undo") == 0)
    {
        UndoRedoManager.undo();
    }
    else if ((_name.compareTo("Round") == 0)
            || (_name.compareTo("Line") == 0)
            || (_name.compareTo("Pie") == 0))
    {
        dP.addShape(_name);
    }
    else if (_name.compareTo("Redo") == 0)
    {
        UndoRedoManager.redo();
    }
}
```

```
ShapeDraw▸UndoRedoManager▸
public static void undo(){
    if (!UndoRedoManager.isUndoStackEmpty())
    {
        UndoRedoManager.addToUndoQueue(
            UndoRedoManager.getUndoString());
        UndoRedoManager.updateUndoQueue();
        int count = UndoRedoManager.
            addUndoCount();
    }
}
```

```
ShapeDraw▸UndoRedoManager▸ *
public static void redo(){
    if (!_redoStack.isEmpty())
    {
        UndoRedoManager.addToRedoQueue(
            UndoRedoManager.getRedoString());
        UndoRedoManager.updateRedoQueue();
        UndoRedoManager.addRedoCount();
    }
}
```

Figure 1 (a) user opens a bubble via the pop-up search box, (b) resulting bubble, (c) user opens definition of two more bubbles side-by-side (automatically grouped); (d) a large working set of bubbles, including a (f) bubble stack of references; (e) an overview is shown in the panning bar; (g) hover preview

dies indicate that programmers spend significant time (re-)navigating through source code and recovering from frequent interruptions. Code Thumbnails [10] and JASPER [11] both attempt to reduce the overhead of navigating through source code by providing visual tools that allow users to exploit spatial memory. CodeThumbnails uses thumbnail displays which have a perceptible structure but which cannot be read, to afford compact intra-file views (by extending scroll bars) and inter-file views (by providing an overview window). Alternatively, JASPER provides views of collections of read-only code fragments, explicitly harvested by the programmer while viewing, that can be spatially arrange in 2D and which are hyperlinked to their original source files. Neither system uses a continuous display or code reflow, although JASPER attempts to address scalability issues by dynamically shrinking font size to fit more fragments to the display window and by providing a button for non-incrementally re-tiling fragments without overlap.

Other research has focused on reducing the cost of navigating to specific code fragment by making working set fragments directly accessible via a list. Many of the techniques pioneered by these tools, *e.g.*, determining working sets based on navigation recording and analysis [12] [13] [14], project histories [15], user input [16] [17], or a degree-of-interest model [18], are complementary to our approach and could be integrated with Code Bubbles.

Outside of IDEs, there have been a number of systems which attempt to provide UIs for gathering working set fragments. The Sandbox [19] supported analytical sensemaking by providing a UI for harvesting information fragments from documents and then arranging and annotating that information in a 2-D space. WinCuts [20] augmented the Windows metaphor to allow users to create live application clippings. Neither, however, provided a continuous display or supported incremental clipping tiling.

More generally, research in windowing UIs dates back decades to when the two dominant classes of window metaphors, tiled and overlapping, were created. The prevailing philosophy is that overlapping windows provide flexibility by conforming to their contents, whereas tiled window displays reduce interaction burden by algorithmically tiling free space [21]. Code Bubbles is thus a hybrid of these two approaches because it combines the free-form layouts of overlapping window managers with automated layout techniques that reduce interactions. Research has also explored virtual extensions to the display surface such as discrete Rooms [22] which are scalable, but require explicit Room/task transitions. Alternatively, Scalable Fabric [23] employs a focus+context technique in which groups of windows representing a task are simultaneously visible and reduced in size as they approach the screen periphery, but this approach is limited by the display screen size.

DESIGN OVERVIEW

Although it is possible to create side-by-side displays of multiple code fragments with conventional UIs, pilot testing with 5 professional developers indicates this is difficult to do in practice for more than several functions even though professional programmers in the study indicated that such displays would be quite valuable [24]. We attribute this apparent contradiction to the fact that conventional UIs, by their nature, make it prohibitively difficult to create side-by-side views of code, for several reasons, including:

- File-based views are generally large by default, requiring multiple interaction steps to concisely display an individual method
- Code in its natively written form leaves significant white space when fit to a rectangular window, and does not readily fit into a compact space
- Modifying a layout of panes or windows takes multiple interactions steps, whereas scrolling or switching panes may take a single step.
- Window layouts are generally limited by the physical size of the display screen.

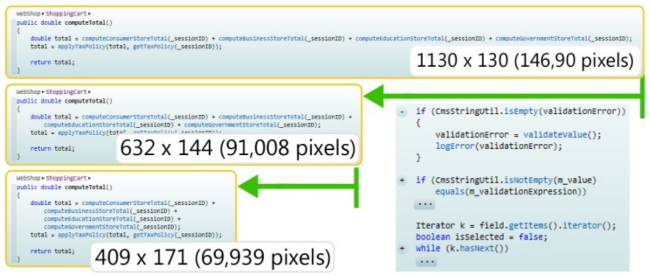
Code Bubbles represents our attempt to adapt the window UI metaphor for code viewing, such that there is neither a penalty for creating side-by-side views nor a loss in efficiency when initially accessing methods.

SPATIALLY-EFFICIENT METHOD BUBBLES

The fundamental design choice of Code Bubbles is to display code, by default, at the granularity of individual methods instead of files. Files, we believe, are a necessary way to communicate with current compilers, but are not the sole or best way to view and edit code. Unlike file displays, bubbles can be automatically sized to tightly fit the method they contain, thus avoiding the many scroll and resize steps required to achieve this effect in IDEs. Many bubbles (11-17 in typical case analysis of large open source applications, JEdit, JForum and ArgoUML) can be shown concurrently since code often consists of short functions [29].

Reflow and vertical elision

To be generally applicable, however, bubble-based displays must also be able to display spatially succinct representations of longer methods in terms of line length (80 characters) and lines of code (100 or more). We thus restrict the a bubble's initial dimension to 55 characters by 40 lines and apply automatic reflow to shorten lines without text clipping, and automatic elision to abbreviate longer functions.



```
WebShop ▸ ShoppingCart ▸
public double computeTotal()
{
    double total = computeConsumerStoreTotal(_sessionID) + computeBusinessStoreTotal(_sessionID) + computeEducationStoreTotal(_sessionID) + computeGovernmentStoreTotal(_sessionID);
    total = applyTaxPolicy(total, getTaxPolicy(_sessionID));

    return total;
}
```

```
WebShop ▸ ShoppingCart ▸
public double computeTotal()
{
    double total = computeConsumerStoreTotal(_sessionID) + computeBusinessStoreTotal(_sessionID) +
        computeEducationStoreTotal(_sessionID) + computeGovernmentStoreTotal(_sessionID);
    total = applyTaxPolicy(tot

    return total;
}
```

```
WebShop ▸ ShoppingCart ▸
public double computeTotal()
{
    double total = computeConsumerStoreTotal(_sessionID) +
        computeBusinessStoreTotal(_sessionID) +
        computeEducationStoreTotal(_sessionID) +
        computeGovernmentStoreTotal(_sessionID);
    total = applyTaxPolicy(total, getTaxPolicy(_sessionID));

    return total;
}
```

```
-  if (CmsStringUtil.isEmpty(validationError))
   {
       validationError = validateValue();
       logError(validationError);
   }

+  if (CmsStringUtil.isNotEmpty(m_value)
       equals(m_validationExpression))
   ...

   Iterator k = field.getItems().iterator();
   boolean isSelected = false;
+  while (k.hasNext())
   ...
```

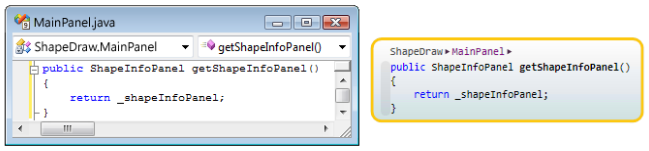
Figure 2 *Reflow can reduce function footprint (left); vertical elision (right)*

Since naïve text wrapping approaches produce source code that programmers may find "unreadable", we use a syntax-aware algorithm that, mimicking the manual reflow strategies used by programmers, aligns wrapped text to commas, parentheses, and other operators (Fig. 2). Reflow operates at the view level, and so does not modify the underlying source code. Code can be viewed in its natively written state if its containing bubble is resized wide enough. This technique is similar to Eclipse's Formatting command, but runs in real-time, reflowing a bubble's text as it is resized, since it operates on individual methods and not entire files. It is important to note that wrapping just one or two lines of code can significantly reduce the overall bounding box area of most methods since each wrapped line adds only one character height to the box's vertical dimension but typically reduces the box's width by numerous character widths.

To handle longer functions, we provide several mechanisms, including the common techniques of vertical scrolling with a scroll wheel and bubble resizing. In addition, for functions of more than 40 lines we automatically elide basic code blocks until the function fits within 40 lines (Fig. 2). If the function is still too long, or if users have selectively expanded some of these blocks, we display a scrollbar.

Minimal bubble decorations

A natural consequence of representing methods as bubbles is that each bubble needs to be a standalone element, capable of supporting interactive manipulation and disclosing its semantic context. We also wanted to avoid window decoration not only because of its spatial bloat but also because it would introduce significant "visual clutter" that would distract programmers "just trying to read" code (Fig. 3). Thus, we minimized explicit bubble decoration to a thin border, which acts as a resize handle like the border of a traditional window. Instead of a title bar, we blend a breadcrumb bar into the top of a bubble to provide the semantic context of the package and class names that contain the method. Clicking the breadcrumb bar provides an alternative to scrolling through a class file to see other methods; when clicked, a list of methods and other items from the class appears; hovering over an item shows a modeless preview bubble, and clicking opens that item in a bubble. Finally, since bubbles lack a title bar with a close button, we provide alternative methods for moving and closing: right-click dragging moves a bubble; middle-clicking a bubble closes it. Standard operations are preserved; left clicking manipulates the text caret, and right-click without dragging opens a context menu. Although these interactions are unfamiliar, they do not override any expected functionality, and could potentially be made self-disclosing [25]. Moreover, we expect programmers will find this UI more efficient than the title bar UI since the target area is the entire bubble, not a small UI element. Indeed, given the ease with which bubbles can be closed, we also display a semi-transparent undo button in place that gradually fades away whenever a bubble is closed; clicking it re-opens the bubble in the same position.



```
public ShapeInfoPanel getShapeInfoPanel()
{
    return _shapeInfoPanel;
}
```

```
ShapeDraw ▸ MainPanel ▸
public ShapeInfoPanel getShapeInfoPanel()
{
    return _shapeInfoPanel;
}
```

Figure 3 Left, MDI child window from Visual Studio, right a Code Bubble

CREATING AND MANAGING 2D BUBBLE LAYOUTS

To make simultaneous views of methods an integral part of programmer workflow we made several design choices that attempt to make simultaneous displays of multiple bubbles a default effect instead of something that the user has to work to achieve. To accomplish this, we adapted existing behaviors, such as "go to definition" and "find all references", implemented a pop-up search box, and implemented a novel Spacer algorithm that incrementally adjusts the placement of bubbles to avoid overlap between bubbles.

Automatically spacing bubbles to avoid overlap

Bubbles are inherently non-overlapping; whenever a bubble is placed such that it overlaps another bubble, a Spacer algorithm is automatically invoked. The Spacer pushes other bubbles out of the way via a smooth animation, while trying to minimize overall bubble movement and preserve spatial adjacency (cues likely to be important for spatial memory). Thus, programmers can move one bubble next to another in a single dragging step that incrementally modifies but does not completely disrupt their bubble layout (Fig. 4).

While optimally bin packing items is NP-hard, we implemented a heuristic recursive algorithm that attempts to find a global minimum in at most 400ms (typically < 100ms); if a solution is not found the "best so far" is used (pseudocode is available [24]). The algorithm works by heuristically generating a set of valid placements (sequences of bubble movements) that contain no overlapping bubbles, and then computing a score, the total Euclidian distance moved of all bubbles in each placement relative to their original position, and finally choosing the placement with the lowest score.

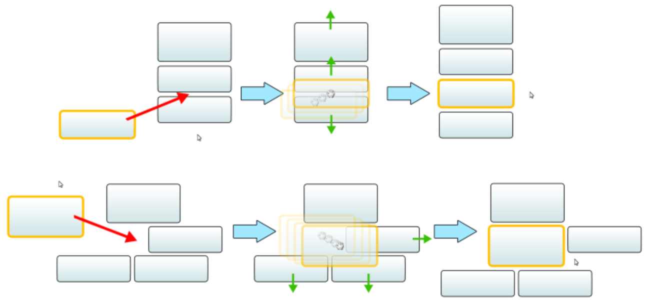


Figure 4 Left, initial configuration, center, user drags bubble into position, right, intersecting bubbles pushed out of the way

A recursive helper function generates a set of possible placements: it begins by marking (to track what has been moved) the user-moved bubble(s); all other bubbles are unmarked. It assigns the "just-moved" set, J, equal to this bubble, and assigns I_j to the set of unmarked bubbles which intersect any bubble(s) in J. If I_j is empty, then the cumulative movement sequence is added to the output set of possible placements (base case). Otherwise, the recursive case of the algorithm generates a set of 4 axis-aligned movements (up, down, left, right) which move i the minimum distance such that i no longer intersects any marked bubbles. Movements that are in the opposite direction to that moved by any bubbles in J that intersect i are pruned (except on the initial call). The 2 movements with the largest Euclidian distances are kept, the rest are pruned. Then all movements for each i are appended to the set of movements for every other i. For each possible combination of movements (1 movement for each i) – pruned to those that when applied do not cause bubbles in I_j to overlap with each other or marked bubbles – it marks and moves (according to the movement combination) all bubbles in I_j, sets J equal to I_j and recursively calls the helper function on the new J and I_j.

The Spacer does not guarantee an optimal space-filling result, nor does it guarantee that all spatial adjacency relationships are maintained; however, its animated, incremental nature produces results that are reasonably predictable and we believe are not likely to seriously disrupt programmers' spatial memory of their code layouts. The Spacer is invoked for user-directed layout changes beyond move, including opening bubbles, resizing bubbles, etc. (see below).

Writing new code

Bubbles are fully editable and automatically resize vertically to accommodate new lines of text typed into a bubble, causing the bubble spacer to push any bubbles below the current bubble out of the way. This process is quite similar to what happens in a conventional text editor; functions below an edited function shift down as new lines of code are entered above them. Once a bubble reaches its default maximum height, it will start to scroll instead of continuing to grow. A unique "budding" event happens when text is entered at the very bottom of the bubble outside the lexical scope of the bubble's contained method – this new text automatically spawns a new bubble that is logically in the same class as its source bubble. Thus, there is no additional overhead for creating a new method – the user just types the new class method at the bottom of any bubble from that class (which can be thought of as a proxy for the class' file). Bubbles for the same method can be opened multiple times; editing one updates the others in real time.

Opening new bubbles

When the context menu option of Open Declaration is chosen for a function call in a method bubble, a new bubble is created for its declaration. We consider potential placements for the new bubble that are adjacent to the source bubble and choose the placement that will cause the spacer algorithm to shift other bubbles the least; the spacer algorithm is automatically invoked for all new bubbles. Newly-created bubbles are highlighted in orange, to draw the users' attention away from any shifting bubbles. A fade-out animation gradually returns the background color to normal. Since bubble layouts can quickly become visually complex after just a few Open Declaration actions, we attempt to improve self-disclosure by drawing arrows, or bubble connections, between function call lines and their corresponding method bubble definitions. Bubble connections have a similar visual appearance to electrical circuit diagrams, and reserve space as needed to reduce overlap. Hovering over a bubble recursively highlights the connections and code lines that lead to it (Fig. 5).

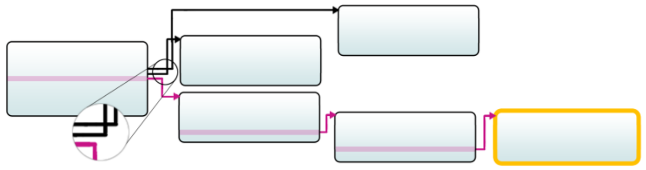


Figure 5 Rectilinear bubble connections illustrate calling sequences. Connections are drawn to avoid overlap (inset).

Users can also search for bubbles by name using a pop-up search box (Fig. 1A) by right-clicking on the background. Initially, a list of all packages and classes is displayed. Users can browse via mouse or keyboard, or use Boolean substring matching similar to that provided by Visual Studio 10 (not publicly available at the time of writing); our implementation separates Boolean search terms by spaces instead of by CamelCase. Hovering over items in the list previews the method as a tooltip (Fig. 1G); pressing Enter or clicking will dismiss the search box and open the method as a bubble, in place. Dragging items from the list does not dismiss the search box, allowing multiple methods to be opened from a single search list. The spacer is invoked incrementally to eliminate overlap as needed. Thus searching for and opening a new method bubble involves equivalent work to searching for and viewing a method in a conventional IDE.

Find All References and Bubble Stacks

Similar to Eclipse and Visual Studio, a Find All References function displays a list of the source lines containing an indicated text string. However our UI is logically a bubble stack (Fig. 1F) comprised of two columns: the source code line with the matched text highlighted, and the name of the containing function. Results are grouped by package, class and then method. Hovering over an item previews a bubble display of the method as a tooltip; clicking expands the item in place as a bubble; clicking the page up/down keys flips through bubbles for each search result. Bubble stacks are hosted inside bubbles, meaning that multiple can be open side-by-side and each participates in bubble spacer layout.

VIRTUALLY EXTENDING SCREEN SPACE

Despite efforts to make bubbles spatially efficient and to provide automatic layout support, some working sets will likely exceed available screen space and some users will want to multi-task between distinct working sets. Thus we considered several virtual screen space extensions. We rejected allowing bubbles to overlap which we felt would burden the programmer with frequent Z-order management decisions in addition to complicating when and how the Spacer should be invoked. Similarly we rejected doing nothing since that would burden the user with explicitly having to decide when to delete bubbles even if they knew they would need them later. We also rejected geometrically scaling bubbles both because that would provide only limited additional space and because in formative evaluations, several programmers commented that they found it hard to read code of different, particularly small, font sizes. The option that remained was to provide virtual screens, however, we also noted that many programmers strongly disliked the discrete Rooms [22] metaphor because it forced them to explicitly and completely change working sets and still did not support a working set larger than the display screen. Thus the design implemented is a large, but not infinite, continuous virtual screen that expands if needed.

2-D, Continuous Workspace

Initially, Code Bubbles displays the center region of a virtual workspace that is 20 times the width of the display and 1.5 times the height. Bubbles can be placed within this view, but when extra space is needed, the view can be panned in 2-D by right-clicking and dragging on the background to reveal additional space (accelerated by a gain factor of 1.5). Thus, rather than deciding to close bubbles to make room, we expect users may choose to work left to right as they add new bubbles and pan or push older bubbles off screen to the left. New working sets can then be created by just placing bubbles in an unused portion of the screen. If bubbles are pushed below the bottom of the screen or above the top, then the virtual workspace is automatically grown in size to accommodate them, although we expect the virtual screen is easiest to use when it is roughly the height of the display since it is harder to lose things when only 1-D left-to-right panning is needed.

To simplify interaction with working sets larger than the physical display, we support a transient zoom feature (Fig. 6). Pressing F9 toggles between a default and a 50% reduced view. A smooth animation, centered on the current viewport is used. While zoomed out, users can move distant bubbles to the current view, or re-arrange groups of bubbles; users can also click on a bubble to zoom back in. To facilitate keyboard-based interaction, each bubble is assigned a non-mnemonic, single-digit alphanumeric overlay; typing the corresponding key zooms back in on that bubble.

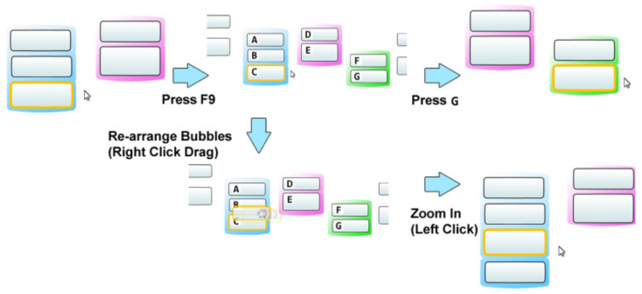


Figure 6 *Zooming out to switch or re-arrange bubbles*

To further support scalability of the virtual display, we provide a panning bar (Fig. 1E) which shows an overview map of the entire workspace, and a location indicator which shows the size and location of the current viewport.

BUBBLE GROUPS

File-based views provide a convenient, albeit rigid, way for programmers to open, identify, and close groups of methods. We offer an additional mechanism, Bubble Groups (Fig. 7), for operating on groups of methods.

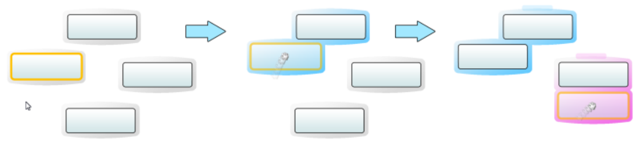


Figure 7 *The user forms two groups by placing bubbles to be adjacent*

Each bubble is surrounded by a semi-transparent halo. When two bubbles are brought within a threshold distance, they join together into a new group. The group is visualized with a colored halo surrounding the bubbles; an unused color from a stored set of colors is used. We used the technical implementation for groups from [26] to display group boundaries. Since collections of related bubbles are typically placed spatially near each other, this automatic visualization alone can provide a useful perceptible structure to working sets. The group halo can be dragged to move the group as a single unit, or double-middle-clicked to close the grouped bubbles in one step (middle-clicking once opens a modeless tooltip and middle clicking again confirms).

Groups can also be named by typing in a title box displayed at the top center of the group, providing a way for users to impose an organizational structure over their working sets. Users can add or remove a bubble from a group simply by bringing it within/outside the threshold distance of the group; groups are recomputed as bubbles are moved. Our prototype naively merges groups when one is moved within threshold of another (sufficient to elicit feedback).

QUALITATIVE EVALUATION

To gain feedback about the utility of Code Bubbles for code understanding and to assess usability considerations, we recruited 14 professional developers (13 male, 1 female, mean age 31.85, SD: 7.13) from the greater Providence, RI area. We advertised broadly with Facebook ads targeting professional developers, recruiting participants from a variety of company sizes and industries. We held two rounds of iterative usability testing with 9 participants in the first round, and 5 in the second (for equipment used, see Quantitative Evaluation). After a pre-questionnaire, participants were introduced to the system, and asked to think aloud. They were given three tasks, similar to those in the quantitative evaluation (below), with the key difference being that the tasks involved writing several new methods.

Usability Results and Discussion

On the whole, developers, with comments like "I could see a ton of people using this," felt Code Bubbles would be very useful to them, and all but one asked when it would be available for them to use at home/work. Developers felt they could see a sizeable number of functions concurrently, in the lab codebase, and that the same would be true of their own code bases. They unanimously asserted that concurrent code views were useful, and that it was "very difficult" to "impossible" to achieve such views with current IDEs.

Developers appeared to perceive value in side-by-side working sets above and beyond reducing navigations; indeed they identified several expected benefits, including: offloading memory; helping them compare and understand functions together; seeing calling relationships, and parameter/return value correspondences; opening a series of functions side-by-side to implement a complex change; and "querying" code to answer specific questions.

They felt the large 2-D virtual workspace was integral for letting them freely "explore" without "getting lost." They felt it made it easier for them to "remember less" since they could easily rescan their working set to refresh their memory, a practice they previously had to support through notes written on paper (one developer commented that he often wrote notes on his hands and arms). Opening new bubbles to the right of existing bubbles kept them from losing their navigational context; even when they had filled the display screen they chose to scroll the virtual display surface instead of closing bubbles. Being able to zoom out with a single key press gave them a "bird's eye view" which they used to "corral" bubbles on a larger scale.

Developers also found value in creating free-form 2-D layouts. They believed that simply juxtaposing related code and arranging bubbles provided them with needed context when analyzing code. In fact, they felt their quick layouts saliently captured enough of their thought processes to improve their ability to multi-task and to recover from interruptions. A shared reaction was that if sections of the panning bar could be named then they could better leverage the virtual space over longer periods of time since previous trains of thought would be better-preserved.

Responses to vertical elision and reflow were more nuanced. One developer found code elision harder to read, while others commented that it made long functions easier to "scan" and read, since they could collapse everything but the particular section of interest. Some developers noted that elided code increased the likelihood that they might overlook important code, but did not anticipate this to be a serious problem. A minority of three developers felt that too much reflow could be distracting and wanted to be able to set a default minimum width for bubbles, however, the majority liked or did not mind reflow, with two even commenting that it made the code easier to read.

Regarding the general UI, developers appreciated the "minimalist" design, allowing them to "focus" on their code, and the use of mouse-buttons to move or close bubbles instead of targeting small widgets; however, several noted that self-disclosure would be helpful. In addition, developers found the spacer algorithm to be relatively intuitive and predictable for avoiding overlap, with comments like "It seems to give precedence to the one you are moving. That makes sense. I like that." They felt that its incremental effect on layout freed them from the "pain" of managing windows and did not move everything "like auto-arrange." One developer felt he might at times want to override the spacer, and suggested a key binding to temporarily disable it.

We were surprised that no developer expressed concern over the absence of files, with some noting that in most cases file contexts were not useful because they tended to be too large necessitating navigation primarily via the package explorer, open declaration, etc. They did, however, note that files would still be useful when an entire class needed to be skimmed or written from scratch, but that we could add a class bubble for such cases, or an easy transition from a bubble to its containing file. In general, however, they felt that bubbles offered better editing opportunities than files because they could view other methods for reference while making edits, or plan for a complex change by opening a set of bubbles to modify.

Developers considered groups to be a multi-purpose organizational tool that they would use because they could create them so easily. They felt groups, in addition to bubble connections, provided needed visual structure to collections of bubbles. Groups and connections made it easy to very quickly locate and keep track of methods, for example, to "follow relationships" or recover spatial orientation and to "find my way back". Several developers favorably compared bubbles and connections to UML diagrams and thought UML integration would be natural and useful. Additionally, groups were perceived as a convenient interface for performing larger-scale rearrangements quickly without disturbing intra-group layouts. Several participants, however, requested an undo feature for changes to groups, and bubble manipulations in general, beyond the undo we provide for close. All but one suggested persisting groups for use later, searching for groups by name or content, and discovering related functions based on group membership.

QUANTITATIVE EVALUATION

In this study, we focus on the twin activities of reading code and navigating code, which when taken together, we term *code understanding*. Developers may choose to make use of additional tools – such as reading check-in statements, using the debugger, using a profiler, etc. to augment their code understanding. However, we argue that these activities build on the core code understanding process, rather than replacing it, in virtually all cases; thus we focus on the more fundamental activity of code understanding. This evaluation investigates the following hypotheses: Code Bubbles users will be able to understand the code more quickly, take advantage of multiple simultaneous bubbles, and should use significantly fewer navigations/minute on average, and fewer repeated navigations/minute on average.

Methodology

We sought to develop a methodology which tested code understanding efficiency with a clearly defined goal. For the purposes of a quantitative study, implementing a new feature is too open-ended and likely to incur significant confounds. On the other hand, a task in which programmers are asked to read code lacks a clearly defined goal; in early pilot tests in which participants were asked to read code to answer specific questions, or identify the cause of specific bugs, they were often unsure of how thorough to be, despite extensive instructions and training tasks.

We therefore chose a task which had a very clearly defined goal that participants could understand and identify with: fixing a bug. Note that the goal of these tasks was *not* to measure bug fixing efficiency, but rather served as a context in which to stimulate code understanding with a clearly-defined goal; to identify and correct the bug necessitated forming an understanding of the features in question. Each bug was designed to require a change to a single, existing line of code, so as to minimize editing/code design as a variable.

Adding additional tool use into the experiment adds additional variables that could confound the results; e.g. we might then be measuring debugger proficiency. Therefore, we restricted developers from using debuggers, trace statements, and other tools and instead asked them to focus on reading the code only. Although this limits the ecological validity of the study, we believe that the benefits in terms of being able to experimentally isolate code understanding behavior from other tools outweighs this limitation.

Participants and Equipment

We recruited 20 graduate and 3rd and 4th-year undergraduate students (19 male, 1 female, mean age 21.95, SD: 2.70) from the Computer Science program of Brown University. We could rely on students having years of Eclipse experience because most computer science courses at Brown use Eclipse. We found in our pilot studies with professional developers that their backgrounds were more varied and that they often use a combination of tools with varying levels of experience with each. Thus, to control for past experience, we used students for the study (consistent with [2]).

Participants reported a mean of 3.85 (SD: 1.72) years of experience with Eclipse, and rated themselves average or higher on a 7-point Likert scale from “beginner” to “expert”; no significant differences existed between conditions. This, combined with the common background in Eclipse experience at Brown, help to address differences in programming ability across conditions as a threat to validity.

All trials used a 24” monitor running at 1920x1200x32-bit, dual-core CPU with 2 GB of RAM, and a GeForce 7300GT graphics card; total cost < $1000 (US). We used an Epiphan VGA2Ethernet hardware video capture system with a VGA splitter to capture the screen without impacting frame rate.

Conditions, Task Context and Tasks

We examined performance in the context of two conditions, a control, Eclipse version 3.4.2 in its default install configuration, and the Code Bubbles prototype with the following features disabled: substring search in the popup search box, groups, and zooming. We disabled these features to reduce the number of independent variables; Eclipse does not have direct parity with these features, and Code Bubbles is usable without them. Both applications were run maximized.

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. ShapeDraw has 32 commands, comprised of 44 classes, 280 methods, and 2,658 code lines (mean of 6.4 lines/function, std. dev. of 13.2); by comparison, ArgoUML, a 150,000-line open source application has a mean function length of 8.7 (SD: 15.5). To control for *a priori* knowledge of API libraries such as Java Swing, we wrapped all non-trivial APIs (but 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. Consistent with [2], code was uncommented because it was unclear how up-to-date/useful the comments should be to be representative.

Users were asked to do one 15-minute training task (Task 0), and two 45-minute tasks (Task 1 and 2). All tasks were designed to not require/benefit significantly from the debugger, to need minimal edits, and to be doable in 30 minutes. Features involved were non-trivial and were designed to represent the scale and complexity of real applications.

For *Task 0*, participants needed to fix a diagram feature that displayed ellipses of various widths and involved 2 classes, 329 code lines, and 22 potentially relevant methods. For *Task 1*, participants were tasked with fixing a bug in the program’s Undo/Redo mechanism. The fix involved understanding how undo/redo actions were stored and applied – participants needed to identify a logic error in the method that stored redo operations. The feature involved 4 classes, 1,017 code lines, and 43 potentially relevant methods. For *Task 2*, participants were tasked with fixing a bug in a bar graph tool which failed to display the correct number of bars on the screen. The fix involved understanding how bar graph data was stored and how the layout system interpreted this data to arrange the bars on the screen. Finding the bug, a logic error in the layout code, involved 4 classes, 897 code lines, and 33 potentially relevant methods.

Experimental Design and Procedure

We used a between-participants design, where participants were randomly given one condition to perform both tasks. After a pre-questionnaire, they were read an introductory statement and given a guided tutorial of six core features of their condition: Package Explorer, Find, Open Declaration, Find All References, Navigate Back/Forward (Eclipse-only), Moving/Closing Bubbles and Panning (Code Bubbles-only), Compile Errors List / Inline Error Squiggle, Save and Run. Mouse and keyboard-based methods for each command were taught (Code Bubbles shortcuts mirrored those taught for Eclipse). For parity, Code Bubbles provided a per-bubble text search UI, and a compile error list to open the appropriate bubbles. These features were more than sufficient to read and understand the code involved in the tasks. Participants were not permitted to use tools to probe the running instance, including debuggers, trace statements, etc.; however they were allowed to use other untaught features that did not probe the running instance, but they were not instructed on their function.

For each task, the program was run and the bug was illustrated and described to each participant. Participants were then told to begin the task by looking at the central handleClick() method which is triggered by all of the event callbacks. This allowed us to simulate an environment in which a programmer was familiar with the high-level architecture of an application, enough to know where toolbar event callbacks occur, but unfamiliar with specific feature

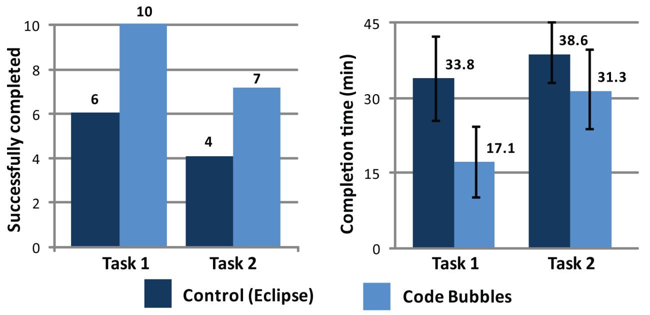


Figure 8 *Tasks successfully completed, and completion time for Tasks 1 and 2 (95% CI)*

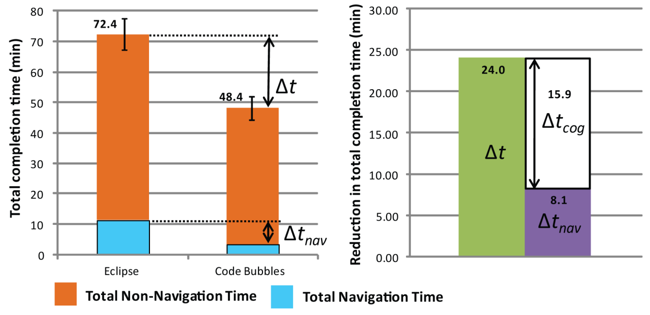


Figure 9 *Reduction in total completion time Δt, reduction in navigation time Δt_{nav}, reduction in completion time not accounted for by navigations Δt_{cog}*

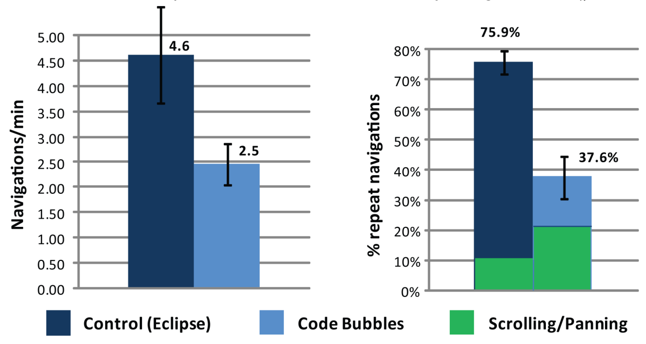


Figure 10 *Navigations/minute (left), repeat navigation rate (right) (95% CI)*

implementations – a routine scenario. It is important to note that developers do not always know where to start, thus placing obvious limitations on the generality of the results.

Participants were given a printed page detailing the task for reference and told the experiment moderator could not assist in completing the tasks but could answer questions about material from the tutorial. Participants were told that they should read and understand the code to identify the bug, and were told that to complete the task they would only need to make and test a change to a single existing line of code (they were permitted to make failed attempts). We asked participants to only modify code when they thought they were entering a solution, so as to control for compile time, auto complete familiarity, etc. In addition, we permitted them to type comments to themselves, make use of a text editor, Notepad, and write notes on paper. Since we wrapped all APIs specific to the task, there was no need to use the Internet and so we did not provide a web browser.

Participants had up to 15 minutes to complete the training task, and up to 45 minutes to complete Tasks 1 and 2. If the participant was not yet complete, they were instructed to stop in order to continue on to the next task. The training task always came first; the order of Tasks 1 and 2 was counter-balanced. Participants were given a break after the first full task; the study took an average of approx. 2 hours. We observed that participants spent their time focused on reading and trying to understand the code, and did not appear to search aimlessly for the bug, or attempt to iteratively probe the code via modifications; therefore we believe the study accurately reflects code understanding behavior.

Results and Analysis

The task completion time and task success results for both tasks are shown in Fig. 8. Code Bubbles (CB) users performed task 1 significantly faster than Eclipse (EC) users ($t_{18} = 2.98,\ p < 0.05$). However, there was no significance in task completion times for the second task ($t_{18} = 1.45, p = 0.164$). Analogous to task completion time, CB users were able to complete task 1 within the allotted time significantly more times than EC users ($\chi_1^2 = 5.2, p < 0.05$), but there were no significant differences for the second task ($\chi_1^2 = 1.0, p = 0.31$). We hypothesize that the reason for the lack of significance for task 2 is that the apparent higher difficulty, when combined with the 45-minute cutoff effectively lowered EC completion times; although the time limit was a tradeoff we made to keep overall experiment time manageable, we hypothesize that a larger time limit could have lead to significance for task 2. Although CB users did not perform significantly faster in completing task 2, they did perform significantly faster than EC in terms of total for both tasks ($t_{18} = 3.83, p < 0.001$). CB users completed both tasks 33.2% faster than EC users (Fig. 9, left). In addition, CB users were able to significantly finish more tasks overall than EC users ($\chi_1^2 = 4.9, p < 0.05$). (The 45-min. time limit places a lower-bound on task completion time for uncompleted tasks).

When users performed both tasks in Code Bubbles and Eclipse, we logged the number of navigations they performed and how much time they spent on navigation interactions[1] (based on screen recording analysis; interaction time for each event was logged to the nearest whole second from the time an interaction began [*e.g.*, moved mouse toward toolbar] to when it completed [*e.g.*, clicked back button]; keyboard actions were logged as 1 sec.). Across both tasks, CB users (Mean: 3.5 min, SD: 1.6) spent significantly less time than EC users (Mean: 11.6 min, SD: 3.08) navigating ($t_{18} = 7.4, p < 0.0001$). This navigation time represents 16% of the total task time spent on both tasks for EC users and 7% of the time for CB users ($t_{18} = 5.47, p < 0.0001$) (we attribute the lower EC navigation time percentage than reported in [2] to the fact that we administered one task at a time, and did not simulate frequent interruptions). In addition, Code Bubbles helped users reduce average navigation time by 68.6%. Examining navigation rate (Fig. 10, left), we see that CB users (Mean: 2.5 nav/min, SD: 0.69) performed significantly fewer navigations per minute than EC users (Mean: 4.6 nav/min, SD: 1.5) across both tasks ($t_{18} = 4.1, p < 0.001$). Of the total number of navigations user performed, we extracted the number of repeat navigations users performed, an indicator of working memory effectiveness. Users of CB performed significantly less repeat navigations per minute (Mean: 0.95 rnav/min, SD: 0.41) than of EC (Mean: 3.51 rnav/min, SD: 1.22) across both tasks ($t_{18} = 6.27, p < 0.0001$). In addition to repeat navigations, EC users exhibited a common behavior in which they "flipped" back and forth between two functions, rapidly (four alternating navigations between two functions), perhaps since they could not see them simultaneously; developers did this an average of 9.2 times (SD: 10.8).

If we consider the total reduction in task completion time for Code Bubbles, Δt = 24.0 min., and compare this with the reduction in time spent on navigation interactions, Δt_{nav} = 8.1 min., we can see that Δt_{nav} only accounts for 33.9% of the time reduction (Fig. 9, right). This surprising result suggests that there was an additional cognitive benefit, Δt_{cog} = 15.9 min., above and beyond the reduction in navigation interaction time, that accounted for the bulk of the performance improvement seen by CB users.

Code Bubbles users all took advantage of having multiple methods open at once; at points sampled 5 minutes before task completion, users had on average 11.0 (SD: 2.87) methods concurrently visible onscreen (does not include partially visible methods), out of a mean total of 17.3 (SD: 5.93) methods open (including partially visible, and offscreen methods). Finally, we logged the number of UI manipulations users did when using Code Bubbles and Eclipse. Specifically, we were interested in the amount of panning CB users did and the amount of scrolling EC users did. These operations are essentially equivalent, given the user interface layout of each tool. We found that there was no significant difference in the number of scrolling/panning operations between CB and EC users ($t_{18} = 0.54, p = 0.6$).

DISCUSSION AND FUTURE WORK

The quantitative evaluation shows that Code Bubbles significantly reduced the amount of time needed to complete the tasks (33.2%), and also the number of successfully completed tasks. In addition, Code Bubbles significantly reduced navigations per minute (46.6%), the amount of time spent actively navigating (68.6%), and the percentage of repeated navigations (50.5%) (arguably wasted interactions). Combined with the qualitative study feedback, we believe that these results confirm our core hypothesis that programmers will be able to understand code faster and with less effort when using concurrently visible working sets of code bubbles, than when using file-based editors, and furthermore, that programmers will qualitatively prefer bubbles for code understanding tasks.

A surprising and unexpected result, however, was that the reduction in navigation time only accounts for 33.9% of the performance improvements seen in Code Bubbles. We hypothesize that the remaining speedup comes from a collection of factors rooted in the limited nature of human working memory. We suspect that limited working memory affects developers in a variety of ways, making it difficult for them to perform several important activities, including: remembering context, comparing and referring back to methods, and re-finding methods when needed. In essence, we believe that developers used Code Bubbles not just to avoid navigation but also to offload their working memory onto concurrent views and spatial arrangements.

We believe that developers used Code Bubbles to rapidly shift their focus across a range of contextually related code fragments, as evidenced by their maintaining an average of 11 complete methods displayed simultaneously on screen. This statistic is notable because we also observed that all participants closed bubbles once they had determined they were not relevant. In addition, we observed that users rarely focused on a single bubble in isolation, but instead appeared to read fragments together as part of a working set, referring back and forth between bubbles as needed. Users also made high-level arrangements of bubbles; for instance, in Task 1, they often opened methods related to Undo and Redo in parallel arrangements to make comparisons. Further, we only noted a single instance in which a Code Bubbles user made a note, whereas Eclipse users made an average of three notes containing important function names, their thoughts about the purpose of a function, etc.

Further evidence that Code Bubbles supported the offloading of working memory comes from the disparity of repeat navigations between Code Bubbles and Eclipse users. On average, 75.9% of all navigations using Eclipse referred back to specific methods they had already seen. In contrast, Code Bubbles users were able to refer to methods that were already on screen. Thus, we noted that only 37.6% of navigations with Code Bubbles were to display previously seen code, and more than half of these were panning operations to see off-screen methods. The remaining were generally attributable to users closing bubbles that they thought would not be relevant later. In addition, Code Bubbles users all frequently juxtaposed bubbles to facilitate direct comparisons. Eclipse users, we believe, attempted to approximate visual juxtapositions by rapidly navigating back and

[1]Navigations logged included Open Declaration, Find All References, Back, Forward, Package Explorer, Find, manipulation of the scrollbar from oe location to another, stopping at and inspecting a function (scrolling through a function did not count), tab switch, pop-up search box, and panning.

forth between methods of interest; a behavior used on average 9.2 times. Four Eclipse users adopted the ingenious and perhaps drastic coping strategy of altering their source file by copying and pasting methods as comments to be adjacent; another pasted methods into Notepad for comparison.

In Code Bubbles, developers also seemed to leverage their spatial memory to glance back to view methods they had just seen, apparently intuiting and adopting the general left-to-right pattern that emerges as new bubbles are displayed. We observed that developers did not always remember the exact location of a bubble, however, in such cases it was interesting to note that they typically remembered the general location, and did not appear to search the entire workspace but intuited the general location. In cases when they needed to pan the screen, participants generally appeared to pan in the right direction to find what they needed. In contrast, Eclipse users relied on their notes which were ineffective in many cases when they had not realized that a method they had encountered would later be important, and thus had not taken any notes. As a result, when they needed to find a method, they often had to re-find it, taking multiple steps. Creating and using working sets appeared to be natural for both the participants in our quantitative and qualitative studies. Since the UI of Code Bubbles is fundamentally a working set, developers did not need to explicitly create one to support a given task since they "got it for free" as part of their normal workflow.

We believe Code Bubbles applies to ecologically valid code bases; professional developers felt a sizeable number of functions would be concurrently viewable in their code, in keeping with the analysis of [29]. Moreover, based on the results, we believe that there is clear value for simultaneous views of code, distinct from the value of tools which facilitate rapid navigation but do not afford concurrent visual comparison via spatial arrangements.

The controlled nature of the tasks places obvious limitations on the generality of the results. Moreover, we do not expect a performance difference would be seen in cases in which developers need to read/edit single methods in isolation, or when the user reads/edits methods in a sequential manner, with minimal referring back to previous methods.

We believe that follow-up studies to further investigate the utility of Code Bubbles in an ecologically valid context are warranted. In addition, bubbles could be applied more broadly to other areas of IDE UI: debugging displays; heterogeneous working sets including documentation, UML nodes, notes, web pages, etc.; interruption recovery and multi-tasking; sharing information; and version management. The bubbles UI might also be applied to other sense-making problems involving information fragments.

CONCLUSION

We have presented Code Bubbles, a novel user interface for seeing and interacting with concurrently visible working sets of code fragments. Qualitative studies indicate that a feature complete IDE based on Code Bubbles would be valuable to professional developers. A quantitative experiment showed significantly improved performance on two code understanding tasks compared to Eclipse, and significantly less time spent navigating. Moreover, we discovered the surprising result that the reduction in performance time could not be attributed to navigation alone, meriting further research into potential cognitive benefits of this approach.

ACKNOWLEDGEMENTS

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.

REFERENCES

1. Erlikh, L. Leveraging Legacy System Dollars for E-Business. I*T Pro*, May/June (2000), 17-23.
2. Ko, A. J. , Myers B, Coblenz, Aung H. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE TSE'06*, 971-987.
3. Plumlee, M. D., Ware C. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. *ToCHI'06*. p179-209.
4. Reiss, Steven. P. The Desert environment. *TSM'99*. p297-342.
5. Nackman, L. R. An overview of Montana. IBM Research '96.
6. Stockton R. Kramer, Nick. The Sheets hypercode editor. 1993.
7. Murphy, G. Kersten M, et al. How are Java software developers using the Eclipse IDE? *IEEE Software* 23, 2006, p76-83.
8. Robillard M. Coelho W et al. How effective developers investiage source code: An exploratory study. *TSE'04*, p 889-903.
9. Sherwood, Kaitlin Duck. Path exploration during code navigation. The University of British Columbia, 2008.
10. DeLine, R, Czerwinski M et al . Code Thumbnails: Using Spatial Memory to Navigate Source Code. *VL/HCC'06*, p11-18.
11. Coblenz M et al. JASPER: an Eclipse plug-in to facilitate software maintenance tasks. *OOPSLA WETeX '06*..
12. Singer, J Elves, R, and Storey. Navtracks: supporting navigation in software. *ICPC'05*, p173-175.
13. Kersten M and Murphy Gail C. Using task context to improve programmer productivity. *SIGSOFT'06/FSE'14*, 1-11.
14. DeLine, R, Czerwinski M et al. Easing program comprehension by sharing navigation data. *VLHCC'05*, p241-248.
15. Cubranic, D, Murphy, G C. Hipikat: recommending pertinant software development artifacts. *ICSE'03*.
16. Robillard, M.and Murphy Gail C. FEAT: a tool for locating, describing, and analyzing concerns in source code. *ICSE'03*. p822-823.
17. Robillard, M et al. ConcernMapper: simple view-based separation of scattered concerns. In *OOPSLAWETex'05*, p65-69.
18. Kersten, M and Murphy Gail C. Mylar: a degree-of-interest model for IDEs. *AOSD '05*, 159-168.
19. Wright, W, Schroh D, Proulx P, Skaburskis A, Cort B . The Sandbox for analysis: concepts and methods. *CHI'06*, p801-810.
20. Tan, D et al. WinCuts: manipulating arbitrary window regions for more effective use of screen space. *CHI'04*, p1525-1528.
21. Bly, S , Rosenber J K. A comparison of tiled and overlapping windows. *CHI'86*, p101-106.
22. Henderson, D. et al. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical UI. *TOG'86*.
23. Robertson, G. Horowitz E, Czerwinski M et al. Scalable Fabric: flexible task management. *AVI'04*, p 85-89.
24. Bragdon, A. Creating Simultaneous Views of Source Code in Contemporary IDEs using Tab Panes and MDI Child Windows: A Pilot Study. TR CS-09-09, Brown Univ. '09.
25. Bragdon, A. et al. GestureBar: improving the approachability of gesture-based interfaces. *CHI'09*, p2269-2278.
26. Watanabe, N. et al. Bubble clusters: an interface for manipulating spatial aggregation of graphical objects. *UIST'07*. p173-182.