Making Programming More Easily in Code Bubbles

Project Report

Yu Li (yli3@cs.brown.edu)

Advisor: Andries van Dam

1. Introduction

Because most of contemporary Integrated development environments(IDEs) are file-based it is difficult to create and maintain a view in which multiple fragments are visible simultaneously. However, Code Bubbles is a novel user interface that is based on collections of lightweight editable fragments, called bubbles, which when grouped together form concurrently visible working sets. The previous 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 report I will describe several visualization cues and advanced features implemented in Code Bubbles, which makes programming easier for developers. In general, those cues and features offer developers critical information of code context and automatic bubble re-arrangement, which save significant amount of time for the users and make them better focus on programming itself. Furthermore, a quantitative evaluation method about comparing the efficiency and the capability of interruption recovery between Eclipse(Paradigm of contemporary IDEs) and Code Bubbles will be proposed.

2. The Bubbles Metaphor

The basis for Code Bubbles is the bubble metaphor described fully in[2]. In this section I will briefly recap the bubble metaphor. The bubbles metaphor represents working set code fragments as individual bubbles that can be freely positioned on the 2-D display surface. In addition, the display surface is treated as 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 to create room for new working set fragments when needed. The bubbles metaphor fundamentally differs from the multi-window UI based in contemporary IDEs, such as Visual Studio or Eclipse. Furthermore, the bubbles have following critical characteristics:

- Bubbles never clip text horizontally, but instead automatically reflow long lines, ensuring that code can be easily read and edited regardless of the dimensions of its bubble.
- Bubbles are not allowed to overlap each other, making groups of bubbles

easier to read since no Z-order management is needed.

- Bubbles have no space-consuming UI decoration, facilitating the simultaneous display of large numbers of bubbles.

3. Bubble Background

3.1 Scenario and Design

Because Code Bubbles is not a file-based Integrated development environment it is easy for programmers to lose the context of entire class. For a instance, several bubbles are opened in current viewport, some of them come from the same class, and the others come from different classes. In order to find the bubbles that come from the same class as the focused bubble, programmers need to read text literally on bread-crumb bar of bubble. If the programmer get interrupted, he might lose the context and need to read text again for resuming programming tasks. Thus, the motivation of using bubble background decoration to highlight important inter-bubble relationships and information is based on the common sense that visualized impression will stay longer.

Specifically, there are two types of bubble background decorations: gradient color and icon image. Each background decoration can indicate either class or package. Users are allowed to enable/disable bubble background decorations and also to change the detailed settings by their own preferences. For example, if gradient background color for class is enabled, all the bubbles which come from the same class will share the same background color; and the bubbles belong to different classes will have different background colors. Figure 3.1 shows the all the possible settings for bubble background. Bubble background only works for code bubble(code fragments bubble) instead of other types of bubbles, for example, note bubble and flag bubble.

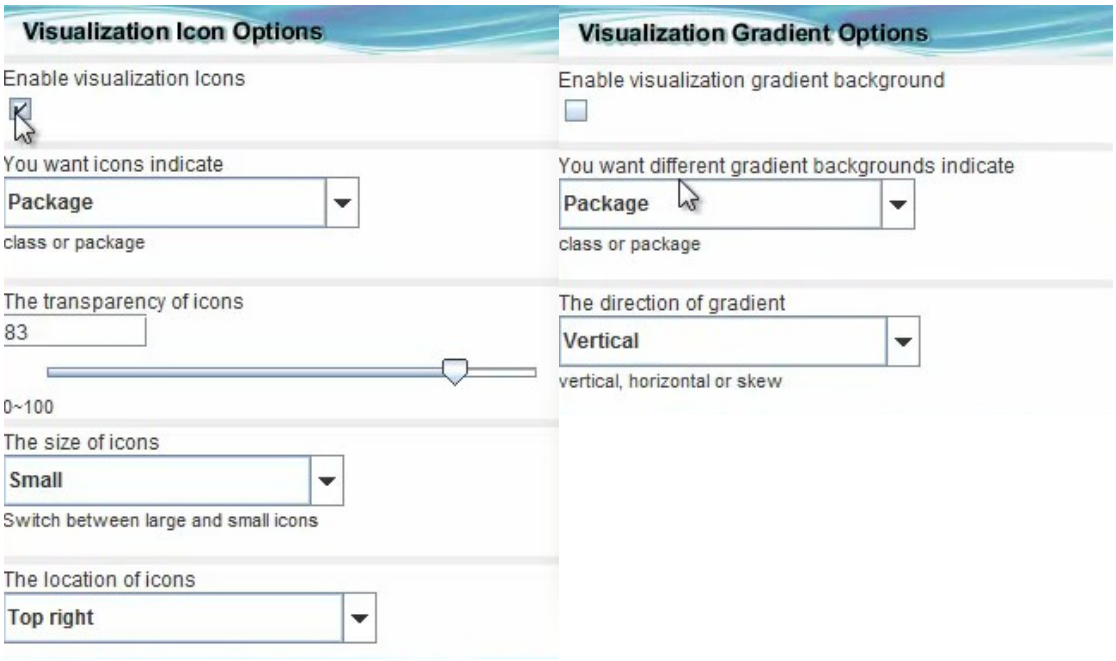

Figure 3.1. Bubble background settings in Option Panel. Bubble background decoration can be either gradient background color or icon, or both. Users are able to assign gradient color or icon to either class or package, which makes a unique color/icon indicates one class/package. For gradient color, direction of gradient can be changed to vertical, horizontal or skew. For icon, the transparency of icon can be changed to any number between 0~100; the size of icons user can be changed to small, medium and large; and also the location of icons can be set to top right, top left, bottom right and bottom left.

For the implementation, I used the HashCode of class name or package name to get the unique color/icon. As we know, many colors are too brighter or too darker to be a background color, so I created a color pool of appropriate background colors for Code Bubbles as well.

3.2 Screenshots

Figure 3.2.1. Screenshot for current viewport. None of the background decorations are enabled at this time. It is difficult to figure out which bubbles are from the same class or package without reading the text on bread-crumb bar of bubble.

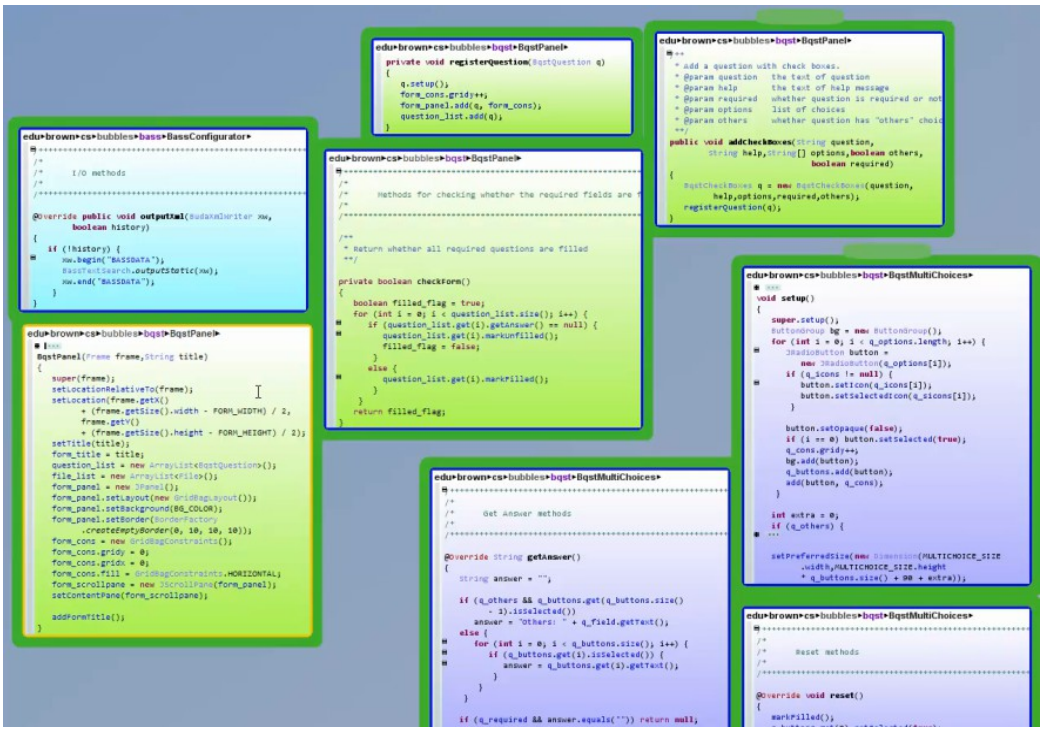

Figure 3.2.2. Screenshot for current viewport. Bubble gradient background color for class is enabled; so the bubbles come from the same class share the same background color.

Figure 3.2.3. Screenshot for current viewport. Both bubble gradient background color for package and background icon for class are enabled; so the bubbles come from the same class have the same small icon displayed on top right corner of bubble; and the bubbles belong to the same package have the same background color.

4. Partial Call Graph

4.1 Scenario and Design

A call graph is a directed graph that represents calling relationships between subroutines in a program. Specifically, each node represents a procedure and each edge(F, G) indicates that procedure F calls procedure G. Call graph is a basic program analysis result that can be used for human understanding of program, or as a basis for further analyses, such as an analysis that tracks the values between procedures. The call graph I discussed here is the static call graph, which intends to represent every possible run of the program. The exact static call graph is undecidable, so static call algorithms are generally overapproximatons. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Code Bubbles offers a decent feature called Package Viewer for programmers to view full static call graph of program, and also provides programmers with several displaying methods. However, in this report I will talk more about partial call graph, which is significantly useful for the larger code base. Partial call graph only shows

calling relationships within certain procedures. For example, developer Queenie wants to know calling relationships involving procedure A within procedures A, B and C; so that the partial call graph won't display edge(A, D) and edge(C, B). Figure 4.1.1 shows a typical partial call graph for *bass.BassNameLocation.createBubble* within the bubbles appeared in Figure 3.2.1. Most of contemporary IDEs are able to generate this kind of call graph.

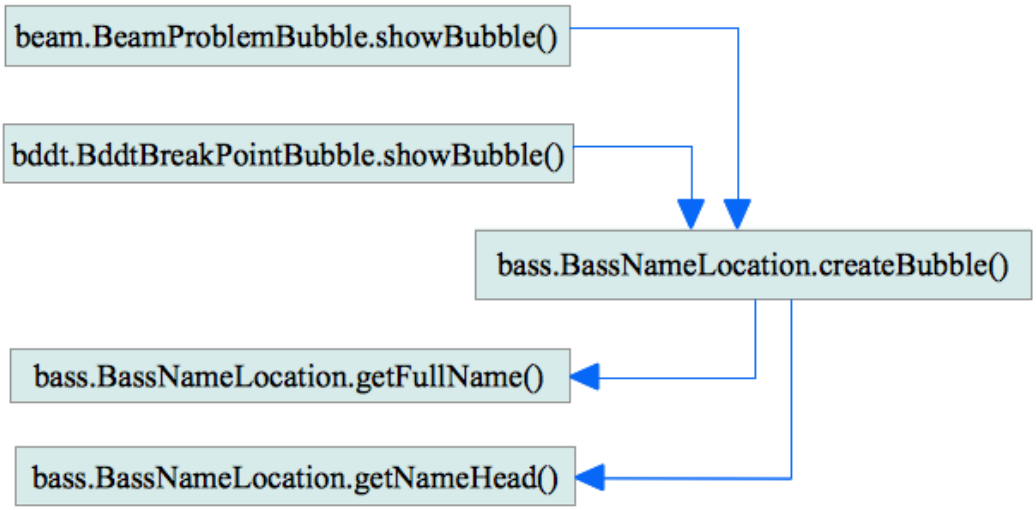

Figure 4.1.1. A typical partial call graph for *bass.BassNameLocation.createBubble* within the bubbles shown in Figure 3.2.1.

Fortunately, the basis for Code Bubbles is bubble metaphor instead of file; so it is easy and straightforward to display calling relationships upon the bubbles by directly drawing arrows between procedures. This approach has the following major advantages:

- Users can browse call graph along with the code; saving time on switching between separate window of call graph and actual code.
- Users can directly find out which line of code results in calling relationship; making easier to understand the code of program.

Specifically, in Code Bubbles programmers can use the keyboard shortcut to toggle the partial call graph; and corresponding inter-bubble edges(arrows) will be drawn on a transparent pane which layers above original bubble area. The partial call graph can be closed by single clicking on the transparent pane. This feature is only working while a valid code bubble has been focused.

4.2 Screenshots

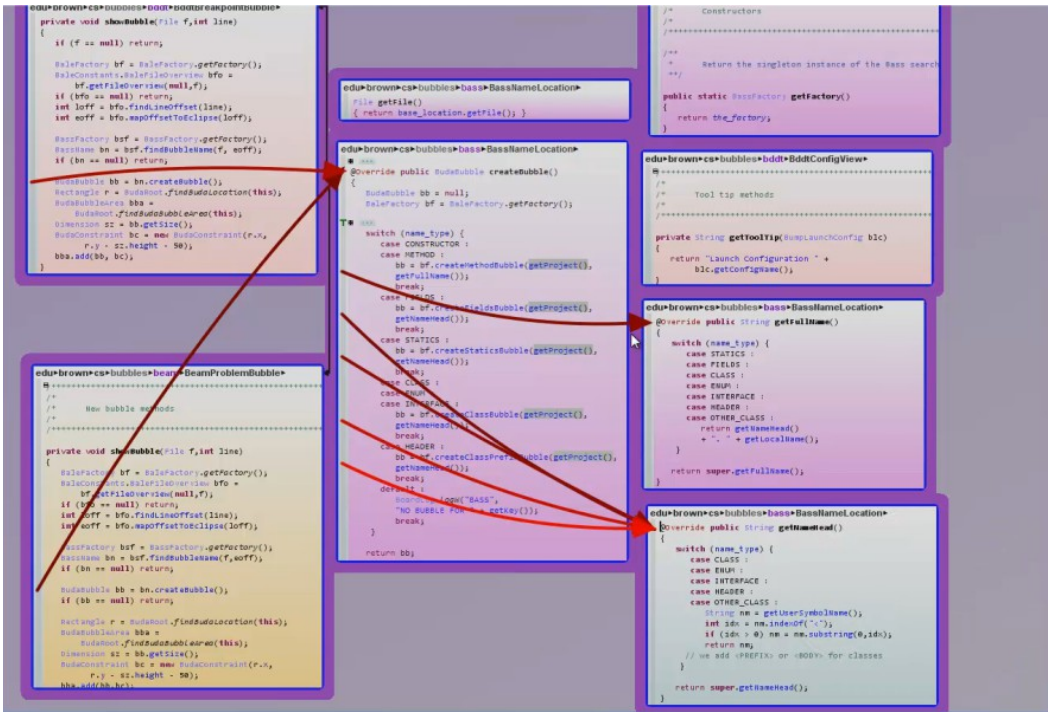

Figure 4.2.1. Screenshot of the partial call graph for method *bass.BassNameLocation.createBubble* within the bubbles shown in current viewport. The start point and end point of arrow indicate the actual line of code where procedure calling happened. Indeed, such representation of call graph is much intuitive and informative than the one shown in Figure 4.1.1.

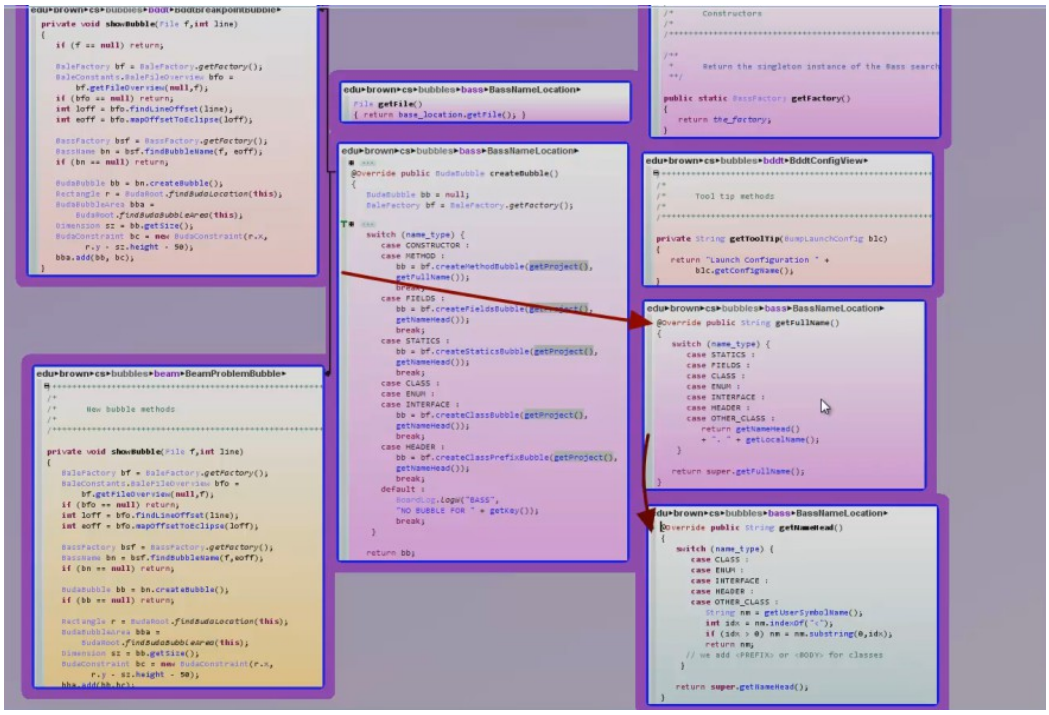

Figure 4.2.1. Screenshot of the partial call graph for method *bass.BassNameLocation.getFullName* within the bubbles shown in current viewport.

5. Automatic Bubbles Re-arrangement

5.1 Motivation

In order to get detailed, qualitative feedback from professional developers, I conducted more than forty phone interviews with about fifteen professional programmers who kept using Code Bubbles as programming tool during last semester. Besides of exploring project and understanding code base, developers also use Code Bubbles quite often for demo purpose during meeting, code review and training; since large number of code fragments can be displayed on the screen at one time in Code Bubbles. As I just mentioned, bubbles in Code Bubbles do not overlap but instead push each other out of the way, which will cause extra empty space between bubbles and make those bubbles disorganized while opening a new bubble. Therefore, the motivation for bubbles re-arrangement is to pack bubbles for programmers automatically; making the bubbles' layout neat, and fitting as many bubbles as possible within current viewport.

5.2 Rectangle Packing

The problem above can be simplified into packing several small rectangles of varying dimensions into a bigger one without them overlapping. Unfortunately, this problem is commonly known as the Bin Packing problem, which is a non resolved combinatorial NP-hard problem. However, I will propose a pretty decent approximation algorithm to the optimal result later.

What we will do is recursively divide the larger rectangle into empty and filled regions. We start off with an empty large rectangle and after inserting one rectangle we will get Figure 5.2.1; then we start to insert the next one, we will get Figure 5.2.2.

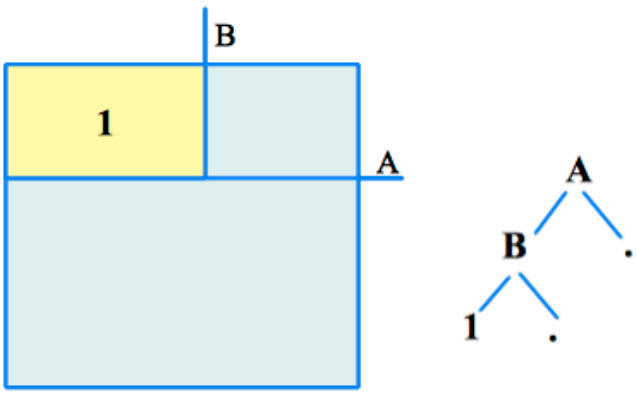

Figure 5.2.1. As image shown, here we've split the bigger rectangle in half by line A

then split the upper half by B and inserted the first rectangle to the left if B.

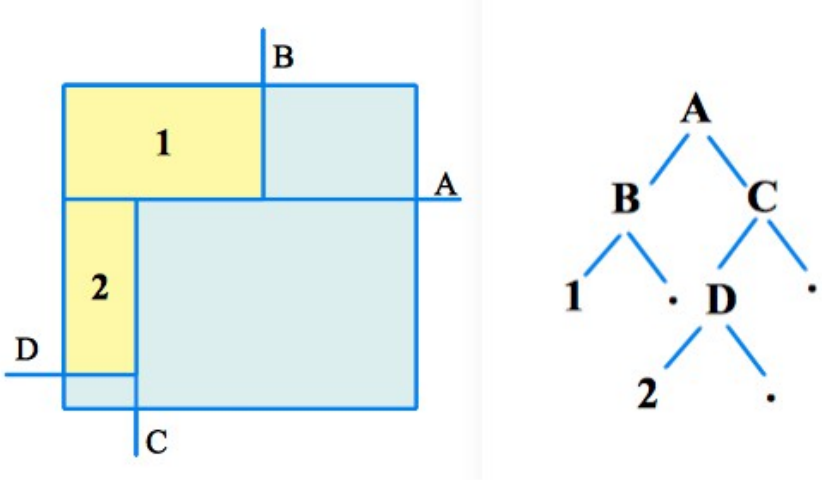

Figure 5.2.2. While inserting the next one, we will check if it can fit above A and if it can we check to see if it can fit left of B, it is full in this case, then right of B. If it fits we split B exactly how we split the original bigger rectangle; otherwise we insert and split below A.

Here it is a part of Java code which implemented above algorithm:

```
01 //help tree node class
02 class PackingNode {
03    int x;
04    int y;
05    int w;
06    int h;
07    PackingNode left = null;
08    PackingNode right = null;
09    boolean used = false;
10
11    //simple constructor
12    PackingNode(int x, int y, int w, int h) {
13       this.x = x;
14       this.y = y;
15       this.w = w;
16       this.h = h;
17    }
18 }
```

```
01 //this is a recursive method that searches appropriate location
   for input rectangles
02 Point findLocation(PackingNode pn, int w, int h) {
03    if(pn.left != null) {
04       Point loc = findLocation (pn.left, w, h);
05       if(loc == null)
06          findLocation(pn.right, w, h);
07    }
08    else {
09       if(pn.used || w>pn.w ||h>pn.h)
10          return null;
11
12       if(w==pn.w && h==pn.h){
13          pn.used = true;
14          return new Point(pn.x, pn.y);
15       }
16
17       pn.left = new PackingNode(pn.x, pn.y, pn.w, pn.h);
18       pn.right = new PackingNode(pn.x, pn.y, pn.w, pn.h);
19
20       if(pn.w-w > pn.h-h){
21          System.out.println("got here?1");
22          pn.left.w = w;
23          pn.right.x = pn.x + w;
24          pn.right.w = pn.w - w;
25       }
26       else {
27          System.out.println("got here?2");
28          pn.left.h = h;
29          pn.right.y = pn.y + h;
30          pn.right.h = pn.h - h;
31       }
32       return findLocation(pn.left, w, h);
33    }
34    return null;
35 }
```

5.3 Screenshots

Figure 5.3.1. Screenshot of Code Bubbles before re-arrangement.

Figure 5.3.2. Screenshot of Code Bubbles after re-arrangement.

6. Evaluation Methodology

6.1 Previous work and Eclipse plugin

In the previous paper, a quantitative evaluation showed that the bubbles metaphor

could improve code understanding performance. In this report, my focus is on how is the efficiency and the capability of interruption recovery in Code Bubbles. As we know, professional developers must frequently resume unfinished programming tasks from where they left off. The interruptions may be due to unexpected requests form co-workers, scheduled meetings, or even extra manual interactions with IDE. Regardless of the source the effects are often the same: When resuming work, developers experience increased time to perform the task, increased errors, increased loss of knowledge, and increased failure to remember perform critical tasks.

Code Bubbles is an attempt for programmers to get less interrupted by IDE itself and reduce their task resuming time by offering various visualization hints. In order to evaluate how well Code Bubbles it is; we want to conduct a quantitative comparison between Code Bubbles and Eclipse with degree-of-interest(DOI) treeview feature. DOI treeview is one of the best cues for resuming interrupted programming tasks mentioned in [3]. Specifically, DOI treeview consists of a treeview of names of the program's parts, namely, its projects, files within projects, which are filtered by a degree-of-interest model over recently visited or edited source code. It also include the ability to decay the DOI model as time passes. Thus, I implemented DOI treeview feature in Eclipse as a plugin. Figure 6.1 shows an Eclipse screenshot with DOI treeview.

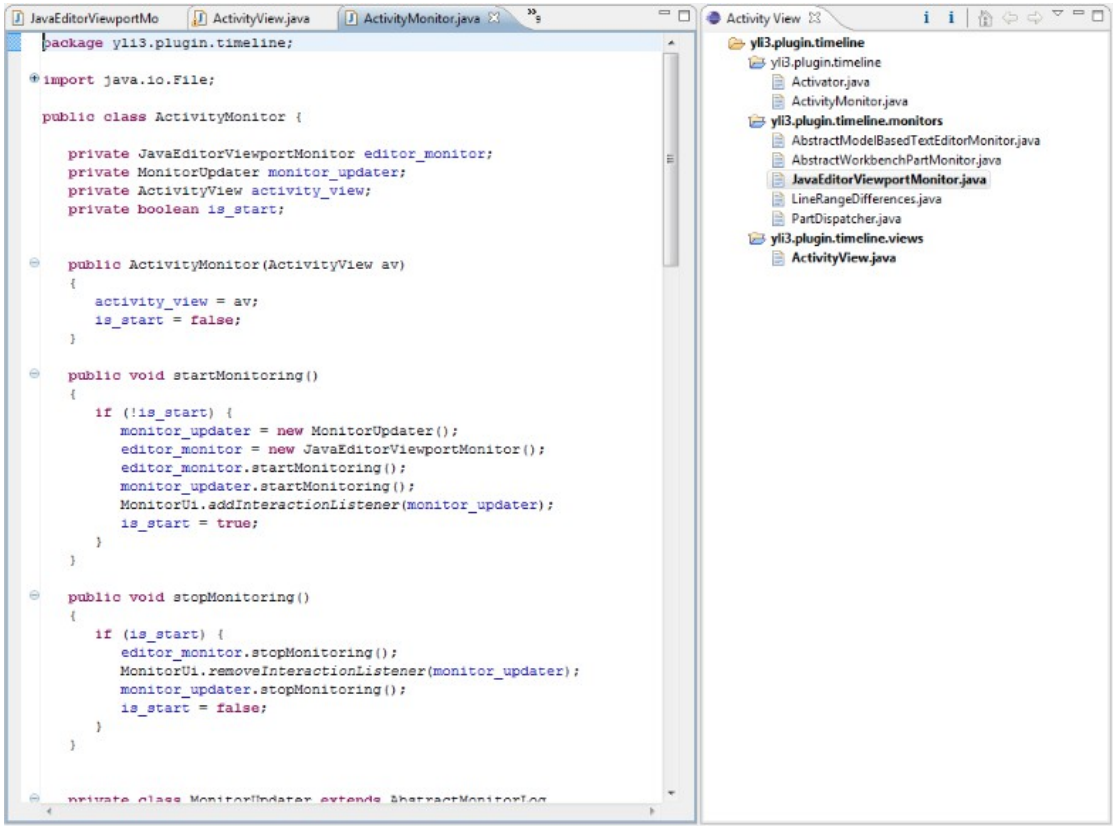

Figure 6.1.1. Screenshot of Eclips with DOI treeview on the right. DOI treeview keeps tracks of recently visited and edited file and also highlights the files have been mostly visited and edited. DOI treeview plugin is built based on Mylyn API.

6.2 Controlled Study

Next step is to run a controlled study to test the effects of two different programming environments(Code Bubbles and Eclipse with DOI treeview) on developers' abilities to recover from task interruption.

Participants: Professional developers. Professional developers are a very demanding customer. They are expert users with significant experience using existing IDE. 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 ideal population for a qualitative study.

Methods and Procedure: We split participants into two groups: one group for Code Bubbles, and the other for Eclipse. In addition, we will use similar application discussed in [3] for controlling interruptions and task switches in the study. When participants work on the pre-prepared simple programming task in Code Bubbles or

Eclipse, the application will automatically interrupted developers by freezing the screen for a few minutes. During the study, we record how many tasks each developer has finished, and how long it will take for finishing each task. Last but not least, we have to choose a time limit for each task that it possible to conduct our experiment within a 2 hour time frame without exhausting participants.

7. Future Work

An advanced feature can be extended from the idea of packing bubble automatically: bubbles can be automatically re-arranged by more intellectual way. This is a complicated problem, since we have to predict programmers' coding behaviors. Here are some possible constraints we can take into account, but how to combine the those constraints appropriately is also tricky:

- Bubbles connected by arrow are more likely to be next to each other.
- Bubbles come from the same class/package are more likely to be together.
- Field bubble is more likely to be on the top of method bubble.
- Bubbles with longest focused time or most editing times are likely to be in a obvious position.

8. Reference

[1] Bragdon, A. el. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. *In Proceedings of the 32nd International Conference on Software Engineering(ICSE 2010)*.

[2] Bragdon, A. el. Code Bubbles: A working Set-based Interface for Code Understanding and Maintenance. *In Proceedings of the 28th International Conference on Human Factors in Computing Systems(CHI 2010).*

[3] Chris Parnin. el. Evaluating Cues for Resuming Interrupted Programming Tasks. *In Proceedings of the 28th International Conference on Human Factors in Computing Systems(CHI 2010).*