Creating Simultaneous Views of Source Code in Contemporary IDEs Using Tab Panes and MDI Child Windows: A Pilot Study

Andrew Bragdon

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-09-09
September 2009

Creating Simultaneous Views of Source Code in Contemporary IDEs Using Tab Panes and MDI Child Windows: A Pilot Study

Andrew Bragdon

Brown University, Department of Computer Science

2009

INTRODUCTION

In this report, we investigate the potential utility of concurrent views of multiple functions side-by-side. First, we present the results of a pilot study with 5 professional developers in which we set out to determine whether or not developers perceive value in such concurrent views, and further how easy or difficult it is to create such views using two approaches in use by contemporary IDEs: tiled panes using Eclipse and overlapping child windows using Visual Studio. Finally, we present the results of an analysis of three open-source applications to see how many functions users might expect to be able to see concurrently in the ecologically valid case.

PILOT STUDY

The goal of this study is to determine whether users perceive value in a seeing multiple functions from different locations in source code simultaneously. Moreover, we wanted to gain qualitative feedback into using existing approaches – tiled panes and overlapping child windows (MDI) – for seeing methods side-by-side. We recruited 5 professional developers (4 male, 1 female, mean age 32.60, SD: 10.26) from the greater Providence, RI area. To get developers from a range of backgrounds, we advertised the study broadly using Internet ads on Facebook.com targeted at professional developers; participants hailed from a variety of company sizes (ranging from startup to large enterprise) and industries (ranging from packaged software development to banks). Developers used a 24" monitor running at 1920x1200x32-bit, Windows Vista, a dual-core CPU with 2GB of RAM and a 150 GB hard disk; total cost < $1000 (US).

After completing a pre-questionnaire, participants were shown Eclipse 3.4.2, a popular IDE, and introduced to its features through a warm-up task. Then they were asked how they felt about tabs in Eclipse and other IDEs; attitudes ranged from neutral, to in the case of three developers, severe dislike, with statements such as "Tabs are a nightmare to work with" and "I personally hate tabs". One developer said that switching between tabs causes you to "lose your context"; another said that he did not "think in terms of files" and that tabs "forced" him to do so. When asked whether they would like to be able to see multiple functions side-by-side, all five participants felt this would be useful. Developers liked the idea of seeing methods from multiple files, and from multiple locations within a file. Developers further said they would want assistance so that the display did not become "cluttered." They felt that they spent most of their time working with multiple functions, and tools supporting this would be useful. They complained of having to navigate between functions repeatedly, and that being able to see all of the functions relevant to a particular task at once could help reduce this. They mentioned that they frequently got lost if the size of the working set was sufficiently large. In addition to reducing navigations, developers felt there were a range of other ways in which seeing methods concurrently would be useful, including: reducing what they had to remember, comparing functions to understand differences, and arranging related items to be adjacent for easy inspection.

Developers were then asked to use Eclipse to create views in which multiple functions are visible side-by-side using its dockable tab panes interface. Developers liked being able to see functions side-by-side, but were displeased with this interface, and unanimously complained about the difficulty involved. Developers felt that it was tedious to position and resize panes, and were further annoyed that once they opened more than two columns, functions no longer fit horizontally, without manually invoking the Formatting command. Developers also complained that adding a new function required them to resize many of the existing panes to accommodate it. Several developers mentioned that they would not do this for more than 2-3 functions because of the level of effort and thought required.

Developers were then asked to use Visual Studio in MDI mode to create views of multiple functions side-by-side using overlapping child windows (one per file). Again, developers liked seeing multiple functions side-by-side but were displeased with the interface, citing different problems. They felt it was "lots of work" to arrange and resize windows in MDI, with one developer comparing it to a "jigsaw puzzle." They felt it was spatially inefficient with

too much peripheral information surrounding each function. They liked the free-form nature of MDI, but did not like managing the Z-order of overlapping windows.

Overall, developers thought it was very helpful to see multiple methods side-by-side, but felt it was prohibitively difficult to achieve this result for more than 2-3 functions. They wanted the system to help reduce or eliminate the tedious and repetitive operations needed to create such working sets. They did not like having to manually drag, scroll, rearrange and resize panes/windows, and although they liked the free-form layout of MDI they did not like having to manage Z-order issues arising from overlapping windows. We further observed in both cases that once the screen had been filled, it became difficult for developers to continue adding additional content, as this necessitated tab switching within specific panes, or changing Z-order in the case of MDI. The file-based nature of the IDEs also caused problems, requiring the user to duplicate the file view to see two locations in the same file.

QUANTITATIVE ANALYSIS OF SIMULTANEOUSLY VISIBLE FUNCTIONS

To build on this result, we further examine how many functions could be seen onscreen concurrently without clipping/panning of the view. For this evaluation we looked at three Java-based open source applications, ArgoUML, a UML designer (349,872 lines of code [LOC]), jEdit, a text editor (102,410 LOC), and jForum, a forum web application (64,033 LOC). Our goal is to analyze how many functions a professional developer working on an ecologically valid code base might expect to be able to see concurrently.

To evaluate this, we used a prototype system which displays single functions in rectilinear windows, each of which has minimal chrome. In addition, the code is reflowed, or wrapped, so that long lines fit within the boundaries of each window; since naïve text wrapping approaches would produce source code that programmers would find "unreadable", we use a syntax-aware algorithm that matches the formatting styles used by programmers when they manually reflow long lines by aligning elements like commas, parentheses, and other operators (similar to the Formatting command in Eclipse).

For each application, we considered the following test cases:

- ***Worst case:*** the longest (largest LOC) functions in the application.
- ***Random case:*** a random selection of application functions.
- ***Typical case:*** all of the functions involved in a specific feature from the application, chosen to be representative.

We compute the following metric 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.

A systematic approach was taken to analyze the mentioned metrics. Code view windows used a specific font (Consolas 8), a package explorer pane was open (others closed), and a 1900x1200 24" display was used. Such a display can be purchased for well under $1000 (US) and thus can be expected to reasonably represent such hardware in the near term. 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, straightforward readability in terms of horizontal width/wrapping (no view was less than 55 characters wide), and minimal scrolling. When no more functions from the test case could fit, we stopped the trial and computed SVF.

		SVF
ArgoUML	*Worst Case*	3
	Random Case	16
	Typical Case	14
jEdit	*Worst Case*	4
	Random Case	16
	Typical Case	14
jForum	*Worst Case*	4
	Random Case	17
	Typical Case	11

Table 1. Simultaneously visible functions in various test cases

Results

Results are presented in Table 1. In real terms, developers would be able to see a sizeable number of functions, 11-17, in the random/typical cases; the worst case provides a lower-bound of 3-4 methods.

These results indicate that with real-world code, one can expect to be able to see a sizeable number of functions side-by-side, 11-17 in the random/typical case analysis, when using a prototype system designed to show such functions concurrently.

CONCLUSION

In conclusion, the results of a pilot study with 5 professional developers indicate that developers perceive significant value in concurrent views of functions spread across different classes and files. Developers encountered significant

difficulty when attempting to create such views using two different approaches, tab panes in Eclispe and overlapping child windows in Visual Studio; developers felt it was prohibitively difficult to create views of more than 2-3 functions. Developers wanted the system to help reduce or eliminate the tedious and repetitive operations needed to create such working sets. Finally, we presented the results of an analysis with three open source applications (ArgoUML, jEdit and jForum) which indicates that developers could expect to see a sizeable number of functions concurrently, 11-17 in the random/typical case analysis, when using a prototype system for displaying such functions.

APPENDIX A: SPACER ALGORITHM DESCRIPTION

This appendix describes the Spacer algorithm, which 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.

A recursive helper function then 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. Marked bubbles will not be moved again. 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).