Fisheye Views of Java Source Code: An Updated LOD Algorithm

J. Louise Finlayson, Chris Mellish, and Judith Masthoff

University of Aberdeen
{lfinlays,cmellish,jmasthof}@csd.abdn.ac.uk

Abstract. One very important aspect of computer programming is reading source code. Whilst this is a relatively simple process for sighted programmers, for blind computer programmers this presents a significant problem. Navigating through and comprehending often thousands of lines of code can be time consuming and difficult. Current development environments have many features that aid the reading of source code for sighted users, however, most of these features are visual in nature and are not translated well into audio by general screen-reader applications. Research has suggested that the use of fisheye views could aid navigation and enhance performance in program comprehension activities for blind programmers. This paper reports the results of a study into creating a better fisheye view of Java source code, by improving the method used to determine each line's 'global importance' or 'Level of Detail' (LOD). The traditional LOD determination method uses only the indentation level of a line to calculate its overall importance. This paper describes the results of the study, and suggests some of the issues which may need to be considered in developing an improved LOD calculation for programming source code.

1 Introduction

Visual systems for displaying and editing program source code use a number of different methods to try to aid program comprehension. Many of these methods act by suppressing unwanted information and highlighting the essential. One such solution, the corollary of a 'zoom' lens allows the user to focus on one specific area in detail whilst ignoring the rest of the information. This approach is not ideal, as noted by George Furnas [5]:

"In such a zoom system local and global information are never available at once. Integration of such information must take place in human memory"

He also discusses the problems involved in a 'two-window' set-up where one window shows the low level minutiae, whilst the other shows less detailed information:

"…this approach has problems, too, basically in understanding the correspondences between the two views. Where does the small view fit in the big picture and what visual features correspond?"

Furnas proposed a novel solution to this problem using a technique known as the 'fisheye' view. A fisheye view [5, 6] is a method used to tailor the level of detail

C. Stephanidis (Ed.): Universal Access in HCI, Part III, HCII 2007, LNCS 4556, pp. 289–298, 2007.

shown at, and around, the user's focal point. This is achieved by calculating the user's likely interest in each line, given their current position, and then displaying or suppressing it accordingly. In this way, the fisheye view of a program is constantly redefined as the user changes their focal point. This method allows the user to view detailed information at their points of interest whilst also displaying the surrounding contextual information.

There is a substantial amount of evidence that supports the assertion that this type of view is more effective than any of the alternative methods mentioned previously. Schaffer et al [7] showed that, compared to the full-zoom method, the fisheye view allowed users to navigate more quickly and retain information about previously navigated spaces more easily. There have also been previous studies which show an increased effectiveness of the fisheye view when compared to other methods such as 'pan-and-zoom' [2, 4]. There has been much interest in the fisheye concept since its creation and at least two program editing environments 'emacs' [8] and 'Jaba' [1] incorporate fisheye lens view features to aid the browsing of program source code.

Our work investigates the use of auditory fisheye overviews to aid navigation and enhance performance in a non-visual program comprehension task. The hypothesis is that providing a glance at the source code in this way will improve the readers' understanding of the program.

2 The Standard Fisheye Algorithm

A fisheye view can be thought of as a 'Degree of Interest' (DOI) filter. The DOI value represents how interested a user is in seeing any particular line, given their current position. The fisheye view defines the DOI of various parts of the structure by taking into account the current focal point, the global importance of each of the lines (LOD) and the structure's proximity to the current focus (see figure 1). In terms of Java code:

- The focal point is the line on which the cursor is currently located.
- The code is treated as a hierarchy, and the distance from the focus (D) is calculated as the path distance between nodes rather than the absolute distance between lines (shown in figure 2).
- The LOD of each line is calculated as a function of its level of indentation, with less-indented lines deemed to be more important than nested structures.

Using these three properties, the DOI function at point x, given the current focus '.' can be defined as:

$$\mathbf{DOI(x|.) = (LOD(x) - D(.,x))} \quad (1)$$

That is, interest increases with global importance, and decreases with distance from the focus. This is the original algorithm developed by Furnas, and used by both Jaba and emacs to generate their fisheye views. Typically, to create the fisheye view, the line x is displayed only if its DOI is above some predetermined threshold. Figure 1, below shows an example of the DOI function calculation for an excerpt of nonsense source code. For each line, LOD is calculated as (indentation level +1) * -1.

For example, line number 11 (at two levels of indentation) is assigned an LOD of (2+1)*-1 = -3.

	Source Code	D(.,x)	LOD(x)	DOI(xl.)
1	XXXXX	3	-1	-4
2	XXXXX	4	-2	-6
3	XXXXX	2	-2	-4
4	XXXXX	1	-3	-4
5	XXXXXX	2	-4	-6
6	XXXXXX	2	-4	-6
7	FOCUS '.'	0	-4	-4
8	XXXXXX	1	-5	-6
9	X	0	-4	-4
10	XXXXXXX	2	-4	-6
11	X	1	-3	-4
12	X	2	-2	-4
13	XXXXXXX	4	-2	-6
14	X	3	-1	-4

Fig. 1. Example showing DOI function calculation for an excerpt of nonsense code

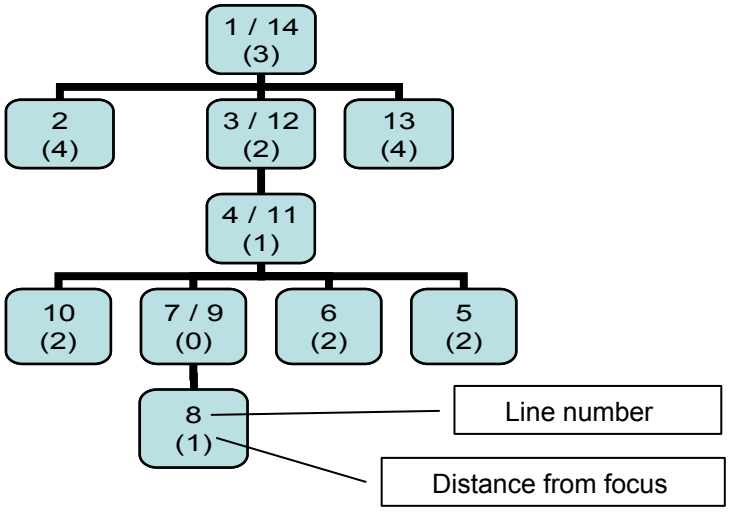

Fig. 2. Path distance calculation for each line of nonsense code (from figure 1)

In our own work, this algorithm was used in an investigation into the use of an auditory fisheye view to aid orientation and enhance performance in a non-visual program comprehension task. The investigation compared two prototype code-reading environments - one incorporating a 'flat' overview of the source code and the other

utilizing a 'fisheye' overview. In all other aspects, the two environments were identical. To create the 'flat' overviews each line was simply represented as its basic construct. For example, the following lines...

```
while (Thread.currentThread() == loopThread){
for (int i = 1; i <= 10; i++){
```

were summarized as a 'WHILE', and a 'FOR' respectively.

To create the 'fisheye' overviews, three levels of detail were represented using two DOI threshold values: A line with a DOI value above the higher threshold was spoken in full, a line with a DOI of less than the lower threshold was represented as a simple tone. Any line with a DOI falling between these two values was rendered as its basic construct (identical to the information displayed in the 'flat' view representation). Both systems used the JAWS for Windows screen-reader software to produce the auditory output.

To evaluate each interface the subjects were required to use one of the systems to navigate through and answer questions on a sample Java program. The time taken to answer each question was recorded, as was the number of correct answers, a log of the user's keystrokes and any comments made. After all the questions had been presented, the subject was asked to complete a NASA TLX workload evaluation to determine how difficult they judged the task to be, and a simple questionnaire about their impressions of the system. This procedure was then repeated with the alternative system, and finally, the subjects were asked to complete a third questionnaire comparing the two systems and indicating their preferences.

The results of this investigation showed a slight trend in favour of the fisheye system (63% of subjects preferred it to the 'flat' overview system and it was judged to be less demanding by the TLX evaluation findings). However, both of these results failed to reach statistical significance. The main finding of the investigation arose from the subjects' comments which revealed that the majority of subjects (88%) felt that the lines that were displayed or suppressed in the fisheye view were not always appropriate, and that important lines were sometimes suppressed whilst trivial lines were displayed in full. This problem is masked (yet not resolved) in some other systems by the heavy use of manual overrides (which allow the user to always suppress or display particular regions regardless of their calculated DOI).

Whilst there is undoubtedly a need for the user to have this manual control, it is essential that the system minimize the amount of work the user has to perform in order to use it, and that the default fisheye view be as appropriate as possible. The large number of negative comments on the information tailoring provided by the default fisheye calculation may indicate that the standard algorithm used is not the most effective in this situation.

The part of the algorithm most likely in need of review is the calculation of each line's LOD. The notion that the importance of a given line can be derived solely from its level of indentation seems to be simplistic at best and does not take into account any information about the type of construct or the content of the line. The study presented in this paper aimed to determine whether there is a more accurate way to assess the global importance of each line than by using the standard algorithm.

It was deemed important that any algorithm developed should only involve information about the source code that is readily available and does not involve an

understanding about the workings of the program. Having a deep understanding about the specifics of the program, and the purpose of every line would undoubtedly provide a very accurate measure of LOD but is an unfeasible requirement for a general algorithm. For this reason, the study concentrated on factors providing structural information such as construct type and indentation. This type of information is easily available to the system and is often already used by programming environments to produce syntax highlighting and other features.

This paper describes the results of the study, and suggests some of the issues which need to be considered in developing an improved LOD calculation for programming source code.

3 Methodology

3.1 Participants

Seventeen subjects took part in the study, all with some experience with the Java programming language. As the experiment did not involve writing code it was not essential that they be expert Java programmers, however, the participants had to be at least familiar with the different types of construct and Java syntax.

3.2 Procedure

Subjects were asked to look at an excerpt of Java code and to consider the overall importance of each line. They then rated each line's importance in contributing to an overall understanding of the program, on a scale of 1 (very unimportant) to 7 (essential). Each subject performed this task five times, with different samples of code. The average degree of importance given to each line, as determined by the subjects, was judged to be its genuine LOD. This LOD value could then be compared to the results of various LOD calculations to find the method that most closely correlated to it.

4 Results

The subjects' Java experience varied from 'use it occasionally' to 'expert'. No significant differences were observed between the answers given by subjects with different levels of experience.

No significant correlation was found between the assessed global importance of a line and its level of indentation ($r = -0.64$, $p>0.05$), see table 1 for the results. This was confirmed by ANOVA, which revealed no significant differences between any of the different levels of indentation ($F(5, 136) = 0.46$, $p>0.5$).

The next factor to be investigated was that of construct type. Earlier work on how best to provide an auditory glance at program source code [3] had revealed that Java could be considered as consisting of six general types of construct, shown below in figure 3.

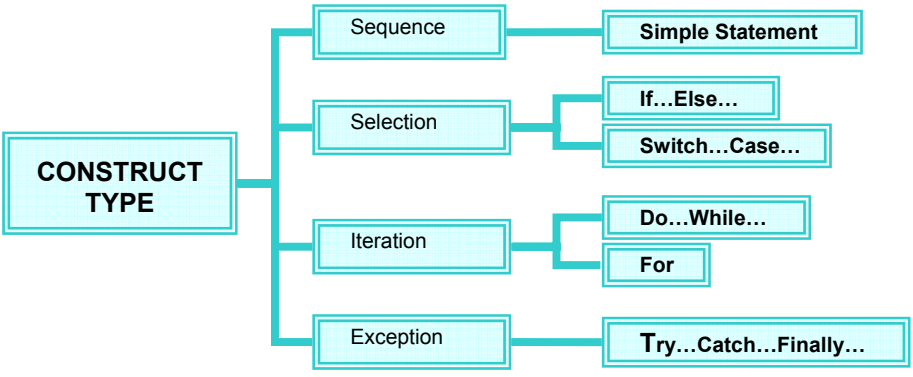

Fig. 3. Original basic Java Construct types

For the purposes of the study, this notion was expanded upon, and three further general categories were identified. These were: Class, Method and End. The Class and Method types include any lines that define either a class or a method respectively, while the End types represent the end of a nested construct, i.e. '}'. This created nine general Java construct types, as shown below:

Type 1 - Class	Type 4 - If...Else...	Type 7 - Try...Catch...Finally
Type 2 - Method	Type 5 - For	Type 8 - Switch...Case...
Type 3 - Simple Statement	Type 6 - Do...While...	Type 9 - End

Of these nine basic types, only seven were represented in the experiment (types 7 and 8 were not included). The mean, assessed Global Importance associated with the remaining seven construct types is shown in table 2. The construct types of the lines of code were compared to their calculated overall importance to determine if any relationship could be seen. The results of the ANOVA revealed that there were significant differences between the groups as a whole ($F(5, 136) = 103.51$, $p<0.0001$).

Further analysis using a multiple range test to determine the source of this variation showed that Type 3 ('simple statements') and Type 9 ('ends') constructs varied significantly from each other, and from the other construct types - with much lower assessed LODs.

With 'simple statement' and 'end' constructs varying significantly from all other types, we looked once more at indentation as a measure of LOD, this time excluding 'end' and 'simple statement' types from the calculation, see table 3 for the results. This time, only five different levels of indentation could be examined as the highest level of indentation can only be filled by 'simple statement' types and was therefore not included in this calculation.

It was shown that there is now a clear, linear correlation between the assessed global importance of a line and its level of indentation ($r = -0.92$, $p<0.01$). This was supported by ANOVA, which revealed there were significant differences between the five groups ($F(4, 36) = 13.77$, $p<0.0001$).

The indentation of 'simple statement' types and 'end' types as a measure of LOD was investigated. In both cases, there were no significant differences observed

between the groups ($p>0.05$), and no semblance of a correlation between a line's indentation level and its genuine LOD.

Type 3 'simple statement' constructs can be further classified according to their specific sub-type. The mean LOD associated with each statement sub-type identified in the experiment can be seen in table 4, below.

These sub-types of statement were compared to their judged level of importance to determine if any relationship could be seen. The results revealed that there were significant differences between the groups as a whole ($F(3, 66) = 13.11$, $p<0.0001$).

Further analysis using a multiple range test to determine the source of this variation showed that all four of these sub-types varied significantly from each other ($p<0.05$).

Table 1. Indentation level as indicator of global importance

	Global Importance	
Indentation Level	*Mean*	*St Dev*
0	4.74	2.06
1	4.51	1.36
2	4.24	0.96
3	4.48	0.92
4	4.28	0.87
5	4.40	0.00

Table 2. Construct type as an indicator of global importance

	Global Importance	
Construct Type	*Mean*	*St Dev*
Type 1 - Class	6.70	0.14
Type 2 - Method	6.03	0.56
Type 3 - Simple Statement	4.25	0.60
Type 4 - If…Else…	5.81	0.53
Type 5 - For	5.12	0.18
Type 6 - Do…While…	5.93	0.23
Type 9 - End	2.86	0.28

Table 3. Indentation level as an indicator of global importance excluding 'end' and 'simple statement' types

	Global Importance	
Indentation Level	*Mean*	*St Dev*
0	6.68	0.11
1	5.89	0.28
2	5.70	0.54
3	5.67	0.58
4	4.40	0.00

Table 4. Sub-class as an indicator of global importance for 'statements'

	Global Importance	
Statement Sub-type	*Mean*	*St Dev*
Method Call	3.72	0.27
Declaration/Assignment	4.73	0.47
Return Statement	4.20	0.28
Other	4.01	0.53

5 Discussion

If the standard fisheye view algorithm were to be substantiated, it would be expected that the indentation of each line would predict the level of importance given to it by the subjects. This did not occur, however, which strongly suggests that the standard LOD prediction using indentation alone is not an accurate method for evaluating the global importance of lines of Java code.

Aside from indentation, the most obvious information about the source code is the type of programming construct. If construct type can predict the LOD of a given line, it would be expected that certain types would attract LOD ratings that differed significantly from other types. This was observed to be true, and highly significant differences were revealed between the LODs of different constructs. The source of this variation was from 'end' types and from simple 'statement' types, which differed significantly both from each other, and from all the other types, with a much lower assessed LOD.

It was hypothesized that these two construct types may have been affecting the original algorithm which used indentation as a measure of LOD. The analysis was repeated, this time excluding the data from the 'end' and 'statement' types. This time, the results were shown to be highly significant, indicating that there were considerable differences between the LOD values assigned to the different levels of indentation. Furthermore, more detailed analysis to determine the nature of this variation revealed a highly significant linear correlation between indentation and genuine LOD.

This evidence suggests that the LOD calculation used to generate the fisheye view for Java source code might be improved with only minor alterations to the original algorithm, by excluding 'end' and 'statement' construct types from the calculation and assigning them lower values for LOD, independent of their level of indentation.

There are however, two issues that need to be considered:

1. This experiment only dealt with six differing levels of indentation (five when excluding simple statements). The mean LOD value for a 'statement' type was 4.25, while, at the most indented level, the mean LOD value for 'other' types was 4.4. Increasing levels of indentation may mean that a degree of overlap arises between these types. Further experimentation with more deeply nested code samples is required to determine whether this 'overlapping' does in fact occur and to establish how to address this issue.

2. The second issue is that 'statement' and 'end' type constructs are by far the most common type of construct encountered in code (in this study they comprised 49% and 22% of all the lines, respectively). It is likely that the LOD of lines of code within these groups may vary, and additional measures may be needed to differentiate between the LOD values of 'simple statement' and 'end' types as a whole.

Analysis showed that neither the LOD of 'end' types nor of 'simple statement' types could be predicted by indentation. Whilst it may be sufficient to assign identical LOD values to all 'end' type code lines, for 'simple statement' types there was a much higher variation between the LODs of individual lines.

The final analysis indicated that the importance of statement types might be associated with their sub-class. It revealed significantly different LOD values between all four statement sub-types ($p<0.05$). This would suggest that statement sub-class may be used to determine LOD, with 'Method Calls' being assigned the lowest LOD, followed by 'Other' types, 'return' statements and lastly by Declaration/ Assignment types.

It is worth noting that, although significant differences were observed between these four statement types, this does not mean that this is necessarily the best method to use to differentiate between different types of statement, or indeed that these are the only four possible sub-types.. Further investigation is needed to look specifically at how best to calculate the LOD of 'simple statement' types to most accurately reflect their genuine LOD.

This division of 'simple statement' types may also lead to a degree of overlap between the LODs of Declaration/Assignment types and the most indented constructs. This would also need to be investigated more fully, particularly with regards to increasingly deeply nested structures, as mentioned previously.

6 Conclusions and Further Work

The results of this study indicate that there may be some truth in the notion that LOD can be determined by the level of indentation. However, they also suggest that it may not be most accurate or effective method of LOD determination for all Java construct types.

The findings of the study strongly suggest that, for 'end' and 'simple statement' types at least, indentation level has no correlation with global importance. Simply by excluding these construct types from the standard LOD calculation we can markedly improve the accuracy of the LOD determination for the remaining types. Further research is needed to determine how much this simple alteration to the LOD determination algorithm can improve the generated fisheye view.

The purpose of this study was to determine whether a more accurate, easily available LOD calculation could be found *for Java source code*. The results we obtained may not apply to other programming languages (particularly non object-oriented languages). For other domains using indent structuring (such as other programming languages, biological taxonomies or decision trees) the general LOD algorithm originally developed by Furnas [5, 6] may indeed prove to be the most accurate method.

Nevertheless, for the purposes of this particular project, it would appear that this updated method of LOD calculation will match more closely the user's opinion of the importance of a given line of code. This may be sufficient to improve the generated fisheye view to such a level whereby the lines displayed by default more closely match the user's expectations with less need for a manual override control.

Acknowledgements. This work was undertaken as part of a project being supported by a studentship awarded by the University of Aberdeen.

References

1. Cockburn, A.: Supporting Tailorable Program Visualisation Through Literate Programming and Fisheye Views. Information and Software Technology 43, 745–758 (2001)
2. Cockburn, A., Smith, M.: Hidden Messages: Evaluating the Efficiency of Code Elision in Program Navigation. Interacting with Computers. The. Interdisciplinary Journal of Human-Computer Interaction 15(3), 387–407 (2003)
3. Finlayson, J.L., Mellish, C.: The AudioView - Providing a Glance at Java Source Code. In: Proceedings of ICAD'05 - Eleventh meeting of the International Conference on Auditory Display (2005)
4. Foltz, P.W., Landauer, T.K., Parker, J.: Design principles for organization and navigation in interactive electronic technical manuals: A review of relevant science and technology (2001) http://www.k-a-t.com/ONR/ONR-HT-IETreview.PDF
5. Furnas, G.W.: The FISHEYE view: a new look at structured files. Bell Laboratories Technical Memorandum #81-11221-9 (1981)
6. Furnas, G.W.: Generalized Fisheye Views, Human Factors in Computing System. In: Proceedings of CHI'86, pp. 16–23 (1986)
7. Schaffer, D., Zuo, Z., Greenberg, S., Bartram, L., Dill, J., Dubs, S., Roseman, M.: Navigating hierarchically clustered networks through fisheye and full-zoom methods. ACM Transactions on Computer-Human Interaction 3(2), 162–188 (1996)
8. Virpioja, S.: Fisheye minor mode for emacs (2005) http://www.niksula.hut.fi/s̃virpioj/fisheye