Navigating Large Source Files Using a Fisheye View

Jakub Dostál [1]

[1] University College Dublin, Ireland
dostal.j@gmail.com

Abstract

As programs grow larger, it becomes more difficult to understand their source code in its entirety. The same is also true for working with source code, which was written by someone other than oneself. A fisheye view is a visualisation technique that displays only information in which the user is currently most interested, while omitting all uninteresting information. This paper presents an implementation of a degree of interest fisheye view of source code and the processes that lead to it, including an extension of a code folding tool for developers using Java Modeling Language. In this paper we present a comparison with other implementations of fisheye view. Also, we suggest a new approach for computing which information is interesting to the user as well as other possible ways for improving the efficiency of a fisheye view.

Keywords: Fisheye view, DOI, information visualisation, Eclipse, code folding

1   Introduction

With computers becoming more pervasive, a large number of applications and programs are being developed every day. These applications are also becoming more complex as the processing power of computers increases. This complexity very often involves teams of developers working with large amounts of source code. The problems that arise from such collaborations are obvious for anyone who has been in such a situation – it can be very difficult to understand a piece of source code that was written by another developer. It is becoming increasingly difficult for developers to be able to understand the source code of their project in its entirety. Moreover, working with an amount of information that increases daily can easily cause information overload, which can in turn cause decrease in work productivity.

There are various ways of dealing with these problems, such as the proposed bubble tree interface for viewing tree structures [Boardman, 2000]. Another is a fisheye view, which is a way of filtering out uninteresting information [Furnas, 1981]. In a basic fisheye view, all elements of source code are assigned an interest value based on their level of detail and their distance from user's focus. The concept of a fisheye view was first introduced by Furnas [Furnas, 1981]. There is another paper that deals directly with a fisheye view applied to a source code file, accompanied with a user study [Jakobsen & Hornbæk, 2006]. This paper presents an alternative to the implementation demonstrated in Jakobsen & Hornbæk along with the description of the design process and decisions. The degree of interest function and its possible improvements are investigated as they are the basis for fisheye view.

2   Related Work

The concept of fisheye view as a visualisation technique was presented by Furnas in 1981 [Furnas, 1981]. His paper presents a fisheye view of structured files and even suggests its possible usability on source code. The proposed fisheye view is a degree of interest fisheye view (i.e., the interest of code is determined by the distance from focus and an inherent interest value for a certain line of code). A definition and an example of

the degree of interest function is provided as well as an example application of fisheye view to tree structures and other hierarchical structures. Moreover, several possible issues were identified, some of which will be discussed in this paper.

In 2006, Jakobsen & Hornbæk published a user study of a fisheye view interface applied to source code. They used a simple editor with fisheye view as a detailed view and a thumbnail view as an overview of the whole file. Their paper is the first user study of a fisheye view applied to source code [Jakobsen & Hornbæk, 2006]. Our work a response to Jakobsen & Hornbæk's paper, and tries to address the issues presented in their paper as well as providing an alternative solution to their approach.

Our implementation of a fisheye view uses elision techniques to fold away uninteresting parts of code. The first paper, which tried to evaluate code elision (here referred to as *folding*) techniques, was written by Cockburn and Smith and published in 2003. They concluded that interfaces that support code elision are more effective than standard linear interface for navigation, especially for large source files [Cockburn & Smith, 2003].

3 Fisheye View

A fisheye view is an information visualisation technique that uses certain principles found when using a 'fisheye lens' in photography. A 'fisheye lens' is an extremely wide angle lens that produces images that show a very high level of detail near the centre of the image. It also shows the surroundings with gradually decreasing level of detail at the same time.

There are two very important concepts associated with fisheye view — focal point and degree of interest. The degree of interest, and the function used to compute it, is discussed in section 5. Focal point is a location in the source file, where the data is deemed most interesting to the user. It is usually in the near the location of the cursor, but as is mentioned in section 5.2.1, there may actually be more than one focal point. In addition to these basic concepts, different implementations of fisheye view use various other concepts and functions. In the following paragraphs we will explain our reasons for making certain decisions about our implementation.

3.1 Design decisions

Apart from using the experience gained from constructing the folding plug-in described in 4.2.3, there were several other reasons why we chose to implement the fisheye view as a structure made of foldable regions.

3.1.1 Main Advantages over Other Implementations

First of all, the main difference is that our fisheye view basically serves as another layer on top of the standard Java editor. This means that all the functionality associated with the editor (e.g. code completion, error hints, warnings, etc.) is retained. Moreover, the folding structure can be unfolded manually by the user and as long as the position of the cursor is not changed, it stays unfolded. In practice it means that if the user is interested in an area of code, which is currently folded, the user can unfold the area without affecting the overall structure. After the user gets back to work (i.e. the cursor position changes), the fisheye view is updated and all the less interesting areas are folded again.

3.1.2 Differences from Jakobsen & Hornbæk's Implementation

In addition to the differences above, there are more differences to the implementation from Jakobsen and Hornbæk. In their paper, Jakobsen and Hornbæk described a fisheye view implemented as a Java editor, which replaced the default Java editor [Jakobsen & Hornbæk, 2006]. In comparison to our implementation, their editor does not support any tools or features from the default editor except for syntax highlighting and line numbering. Also, in addition to the normal editing window, there is a thumbnail overview of the whole source file, while our implementation uses only the fisheye view. This way, the fisheye view is more flexible as the user can use other tools and views available in the default edito to get additional information about the source file.

Also, our implementation addresses several problems outlined in their paper. Jakobsen & Hornbæk state that the static size of focused area sometimes confused users because it was difficult to see any semantic relation between the focused area and user's focus [Jakobsen & Hornbæk, 2006]. We adopt the proposed solution of using the cursor position to determine the focal point. In addition, the size of the focused area is variable

because the focused area always consists of variable size unfolded regions that always well defined to try to avoid confusion about the focused area. Our solutions to other problems mentioned in Jakobsen & Hornbæk are discussed below.

4 A Process Leading to the Implementation of Fisheye View

Our fisheye view is implemented as a plug-in for Eclipse IDE [1]. Eclipse is a well-known software development environment. It contains many tools that should help ease the development process, for example code completion, various overviews, integrated version control system, etc. It also contains tools that allow users to fold away sections of code.

However, before actually implementing the fisheye view using the folding functionality of Eclipse, it was necessary to explore the possibilities available in Eclipse. In order to acquire a deep understanding of how code folding is implemented and how to build plug-ins into Eclipse, we chose to address a problem with JML annotations in the current implementation of code folding.

4.1 Folding in Eclipse

For a number of releases, Eclipse has folding facilities. These are code elision techniques that allow the user to choose, which information is displayed and which is not.

4.1.1 Description of the facilities

The default implementation supports folding of certain areas of Java source code, namely comment blocks, Javadoc blocks, header comments, inner types, members and import containers. There are alternative implementations of folding, available as third party plug-ins, allowing folding of a much broader spectrum of Java elements including blocks of line comments, conditional statements and loops.

4.2 JML Folding Plug-in

After considering several options, we decided to create a plug-in that would solve the problems with JML annotations described in 4.2.2 in Eclipse IDE both because of the experience with folding that we would gain and because of the benefit for developers using JML.

4.2.1 Java Modeling Language (JML)

The acronym JML stands for 'Java Modeling Language'[2]. JML is a formal specification language for Java, which is used by various tools to extensively test the behaviour and correctness of Java modules. In order to describe their behaviour, JML annotations are used. These annotations are Java line or block comments written in a special format (see figure 1 for examples).

Both command line tools as well as tools integrated into IDEs are available. There is an extended static checker (ESC/Java2) obtainable both as a command line tool [3] and an Eclipse plug-in [4]. A set of tools integrating JML into Eclipse is also available from their website [5].

```
// Normal line comment        //@ JML line annotation

/*                            /*@
 * Normal block comment        @ JML block annotation
 */                            @*/
```

Figure 1: Comparison of normal Java comments and JML annotations

4.2.2 Problems with JML annotations in Eclipse

Unfortunately, JML was not supported in Eclipse was not until recently, when several plug-ins became available. And even now the folding facilities, even with third party plug-ins ignore JML and treat JML annotations as normal comments. This means that the choice for a developer, who uses both JML and the folding facilities to fold away the code automatically, is limited to either folding away both JML and all comments, or neither of the two.

4.2.3   The Plug-in

In order to solve the problem, the new folding plug-in was developed. This plug-in was based on the Coffee-Bytes Code Folding Plug-in [6] that extends the folding facilities of Eclipse. Our version of the plug-in makes a correct distinction between normal comments (be it comment lines or comment blocks) and JML annotations and lets the user automatically collapse the two types of comments independently.

5   Determining User's Interest

Determining which information is interesting to the user is essential for the functioning of the fisheye view as the interface provides an overview of the source file. It also provides as much detail as possible for data that is currently interesting to the user. To determine which information is interesting to the user, a degree of interest (DOI) function is used.

5.1   Basic DOI Function

The basic degree of interest function consists of two components[Furnas, 1981]. The first component is the level of detail of a certain element. In Furnas's paper, has been the level of indentation of the element, but we are using the hierarchical structure provided by Eclipse. This component represents an inherent interest value of the element (by default, classes are more important than methods, which are more important than local variables, etc.). Using this information, we construct a tree structure.

The tree structure is then used to compute the second component — the distance from the focal point. First, the user's focus is found and translated into a position in the tree structure. Then the distance from the focal point is computed for every node in the tree. The distance represents the assumption that parts of the source file that are further away from the focal point are less interesting to the user than parts that are closer.

This, however, is only the basic function. In order to make the fisheye view as effective as possible, we propose that other sources of information are used to refine the interest values further.

5.2   Enriching Available Information

In their paper, Jakobsen & Hornbæk identified a number of problems (e.g. what to do in a situation when there are a lot of elements with the same DOI?) with fisheye view [Jakobsen & Hornbæk, 2006]. Several problems were also identified in Furnas's paper [Furnas, 1981]. The majority of the problems have to do with the DOI function. Having more information about user's interest, focus, past choices and the source code itself could have a significant influence on the efficiency of fisheye view compared to the basic DOI function described above.

5.2.1   Program Call-Graph

Eclipse IDE contains information about the program call-graph for the current project. A call-graph is a graph that represents the connections among various elements of the program, e.g. method calls and access to variables. This information can then be used to find areas of high interest that are not necessarily near the current focal point. Using call-graph as the only additional input to the DOI function is not very efficient as this only increases the amount of information that has to be in focus, meaning it is more difficult to maintain the balance between the overall structure and the detailed view of the source file.

On the other hand, using the call-graph is a way of addressing a question from Furnas's paper [Furnas, 1981] about the possible advantage of having more than one focal point to provide more information about the main focal point, where the secondary focal point could be generated automatically.

5.2.2   Source Code Annotations

Another problem, identified in Jakobsen & Hornbæk's paper, is caused by the structure of the source code itself. Most Java source files contain at least one constructor and several methods. In large classes the number of methods and constructors can be quite high. This means that there are numerous elements with the same or very similar degree of interest. This creates a significant problem as it makes the fisheye view very

inefficient.

In order to solve this problem, we propose to annotate the source code to divide the regions into meaningful and manageable structure. These annotations are similar to #region annotation in C# [Solis, 2007]. In addition to the keyword '@region', to define the regions, we propose to use BON [Waldén & Nerson, 1994] keywords and several others (e.g. for constructors) to mark regions of code representing different areas.

5.2.3 Context Aware Tools/Plug-ins

Another source of information for the DOI function is one of the many Eclipse plug-ins available. The Mylyn plug-in [7] is an example of such a plug-in. Mylyn is a context aware task-centred user interface plug-in that can automatically find information relevant to the current task. Using this kind of context information helps with construction of a fisheye view with multiple focal points and also with the precision of the DOI function.

6 Future work

As the current implementation of the fisheye view is relatively primitive in terms of the richness of its DOI function, one of the most important pieces of work to be done is integration of additional sources of contextual information into the fisheye view.

After that, it is important to measure the actual effectiveness and usability of the fisheye view, which will require a user study to be carried out. A comparison is needed not only between the standard linear interface and this version of fisheye view, but also among this fisheye view implementations.It is important to determine, which parts of the fisheye view are actually the most important for usability and which implementation is more effective. It would also be useful to evaluate each source of additional information to see which ones provide the most interesting and important information.

7 Conclusions

The project presented in this paper was a process, during which several important design decisions were made. First, the JML folding plug-in was constructed by extending an already-existing plug-in, which proved to be tricky as the code was not completely stable. Using the experience gained from the JML folding plug-in used to fold away code, we decided to build a fisheye view of source code with very different properties to previous implementations. As a basis for the fisheye view, we used the default implementation of folding, placing code stability above extended functionality. The experience from the JML plug-in makes it possible to add the remaining functionality later.

We managed to build a functional fisheye view. Even though this version was quite simple, it is already had several advantages over other implementations. We believe that when all the proposed functionality is implemented, our implementation of a fisheye view will be more efficient than previous implementations and will the importance of context awareness for this interface. The performance of this interface remains to be tested in a user study.

References

[Furnas, 1981] G. W. Furnas (1981). *The FISHEYE view: A new look at structured files.* Bell Laboratories Technical Memorandum #81-11221-9.

[Jakobsen & Hornbæk, 2006] Mikkel Rønne Jakobsen & Kasper Hornbæk (2006). Evaluating a Fisheye View of Source Code. In *CHI 2006 Proceedings – Understanding Programs & Interfaces,* pp. 377-386.

[Boardman, 2000] Richard Boardman (2000). Bubble Trees: The Visualization of Hierarchical Information Structures. In *CHI '00 extended abstracts on Human factors in computing systems*. pp. 315-316.

[Weiser, 1981] Mark, Weiser (1981). Program slicing. In *Proceedings of the 5th international conference on Software engineering*. pp. 439-449.

[Waldén & Nerson, 1994] Kim Waldén, Jean-Marc Nerson (1994). *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems.*
http://www.cs.caltech.edu/~cs141/2002/resources/WaldenNerson94.pdf

[Cockburn & Smith, 2003] Andy Cockburn and Matthew Smith (2003). Hidden Messages: evaluating the efficiency of code elision in program navigation. In *Interacting with Computers* 15: pp. 387-407.

[Solis, 2007] Daniel Solis (2007). Preprocessor Directives. In *Illustrated C# 2005*. Apress. pp. 503-514.

Websites

[1] http://www.eclipse.org
[2] http://www.jmlspecs.org
[3] http://secure.ucd.ie/products/opensource/ESCJava2/
[4] http://sort.ucd.ie/projects/escjava-eclipse/
[5] http://jmleclipse.projects.cis.ksu.edu/
[6] http://www.coffee-bytes.com/platform_support
[7] http://www.eclipse.org/mylyn/

Source code and plug-ins described in this paper
http://sort.ucd.ie/projects/grok

Appendix 1: A Comparison of Linear and Fisheye View

628
lines of code

50
lines of code

Degree of Interest
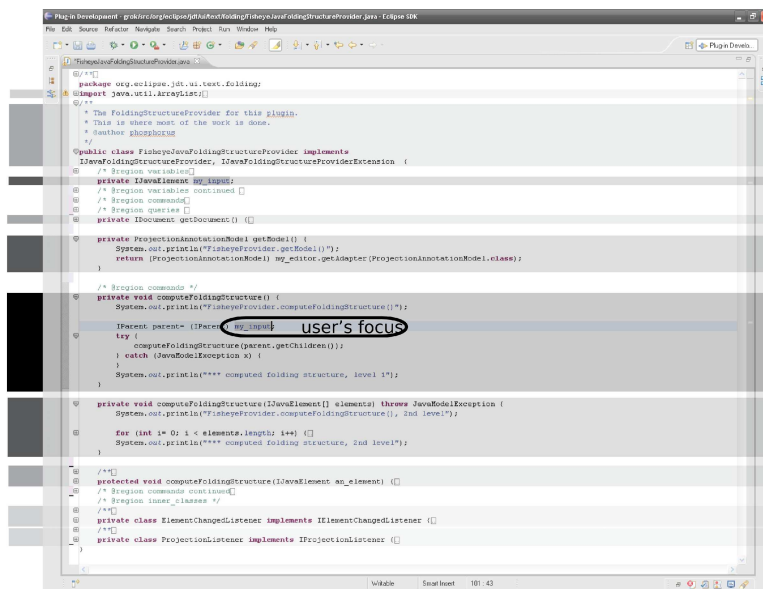
Very High | High | Medium | Low | Very Low

user's focus