An evaluation of the inline source code exploration technique

Michael Desmond & Chris Exton

Department of computer science & information systems
University of Limerick
michael.desmond@ul.ie, chris.exton@u.ie

Keywords: disorientation, source code, exploration, inline, navigation.

Abstract

The exploration of source code in modern integrated development environments can lead to disorientation problems due to a lack of visible exploration context as the programmer moves between successive source code displays.

Inline source code exploration is a technology which facilitates the exploration of source code *in context*. In contrast to explicitly navigating between isolated displays of source code, the programmer fluidly introduces related source code declarations into the context of a primary or focal source code document. The inline approach provides an explicit representation of exploration context between successive source code locations, provides support for the pursuit of exploratory digressions, and allows the programmer to view multiple related source code locations simultaneously with minimal interface adjustment.

In this paper we introduce inline source code exploration and describe a user experiment designed to evaluate the effectiveness of the technique at reducing the level of disorientation experienced by programmers during source code exploration activities.

1. Introduction

Source code exploration is a core and pervasive aspect of the software engineering process. Prior to and during software development and maintenance activities, programmers generally spend a considerable portion of their productive time exploring existing source code in order to identify and comprehend those areas of a system pertinent to their tasks (Singer et al 1997; Ko et al 2005).

Source code manifests a dense, non-linear information space composed of inter and intra related text based source code documents. Structural phenomena such as control flow scatter (Chu-Carrol et al 2003), the use of abstraction, and failure to adequately modularize concerns often leads to a scenario where the portions of source code implementing coherent system operations and features are fragmented and dispersed over a number of source code documents. As such, source code exploration is often characterized by frequent switching between source code displays in order to correlate information and synthesize an overall picture of system implementation across multiple disjoint source code locations.

However, modern integrated development environments (IDEs), such as Eclipse (Eclipse.org 2009), have adopted a user interface design in which the programmer is *effectively* limited to examining a single source code location or 'display' at any moment during exploration activities (De Alwiss & Murphy 2006). Furthermore, there is generally no continuity of exploration context from one display to the next and little or no support for gaining an overview of the exploration task in order to remain oriented. Essentially source code is explored as a sequence of isolated or 'perceptually independent' source code displays which can induce a state of programmer disorientation (De Alwiss & Murphy 2006).

2. Disorientation

Disorientation refers to a sense of *mental lostness* which users experience when browsing or exploring large information spaces. The phenomenon has been identified and studied in a variety of domains such as hypertext systems (Conklin 1987) (Foss 1989) (Kim & Hirtle 1995), spread sheets (Watts-Perotti & Woods 1999) and integrated development environments (De Alwiss & Murphy 2006; Janzen & De Volder 2003).

De Alwiss & Murphy 2006 summarize the phenomenon as when a programmer "loses the context or relevancy of their recent actions to their overall goal".

At the root of disorientation in the computer medium is what Watts-Perotti & Woods (1999) describe as 'the navigation phenomenon'. The information space or 'virtual data field' maintained in a computer system is typically far larger, in a spatial sense, than the physical display space available to the user (the screen real estate). This characteristic is referred to as the "keyhole property" (Woods & Watts 1997). Working on a large information space via a keyhole type display, it is generally impossible for the user to examine all of the information required for a given task simultaneously. Instead the user needs to decide which portions or parts of the information space to call up and examine in a sequential manner. The decisions and actions which drive this process form the essence of navigation in the computer medium.

However because the keyhole display restricts examination to a small portion of the information space at any particular moment, it is often difficult for the user to find and remember information and synthesize a coherent overall picture from information scattered throughout the information space. Typical problems that users encounter are 'getting lost' or disoriented in the display space, where the user is unable to determine where they are and the relevance of what they are examining, as display thrashing (Henderson & Card 1986) where the user has to repeatedly switch between related displays in order to correlate information and as interface management where the user needs to expend additional concentration on interface adjustment and manipulation activities (Watts-Perotti & Woods 1999).

The disorientation experienced by programmers during source code exploration activities may be summarized as problems related to maintaining context and orientation, managing digressions and synthesizing information from related displays (De Alwiss & Murphy 2006).

- **Maintaining context and orientation**

  When exploring source code in an IDE the programmer is effectively limited to examining a small portion of source code at a time due to the limited viewport made available by the source code editor display. However this small portion is generally part of a broader exploration history and context associated with the programmer's current task. Context is important for way finding activities and understanding the meaning and relevance of the current source code location (Storey et al 1999). Because context is not explicitly represented by the IDE the programmer needs to maintain a representation of the necessary context in memory. Loss of context commonly occurs resulting in the programmer getting lost in the source code, unable to remember how or why they arrived at the current source code location and its relevance to their overall goal (De Alwiss & Murphy 2006).

- **Managing exploratory digressions**

  An exploratory digression occurs when a programmer temporarily suspends their primary exploration goal to pursue (or is distracted by) a related digression or side path. This digression may in turn spawn further digressions (embedded digressions (Foss 1989)) and eventually the primary goal may be forgotten. Because digressions are not explicitly recorded by the IDE and the original context of the digression is not maintained, it is easy for the programmer to forget their original goal or to fail to return from a pursued digression.

- **Synthesizing information**

  The fragmented nature of source code means that it is often necessary to consider information from a number of related source code locations and synthesize this information an overall picture or mental model (Chu-Carrol et al 03). Because only a small portion of code is visible at a time the programmer may need to repeatedly switch or flip between related source code displays in order gain the necessary overview and interpretative context. This behaviour is known as thrashing (Henderson & Card 1986) and requires the programmer to concentrate on interface manipulation activities and maintain additional information in working memory as they flip between related displays.

To reduce the incidence of programmer disorientation and generally alleviate the mental burden on programmers during source code exploration activities, the IDE needs to provide support for a visible representation of exploration context, recording digressions and simultaneously examining related portions of source code with minimal interface adjustment. This will then allow the programmer to focus additional mental effort on the primary task of examining and comprehending the source code.

3. Inline source code exploration

One approach to the problem of disorientation during source code exploration activities is inline source code exploration (Desmond et al 2006). Inline exploration is a technique for exploring source code in context. The essential premise is that instead of explicitly navigating between individual displays of source code, resulting in a continuous replacement of visible content with the associated loss of context, the programmer progressively introduces related portions of source code, inline, into the context of a focal source code document. The inline exploration approach only applies to situations where the programmer navigates from one source code location to related source code location via a reference contained in the source code display.

Inline exploration facilitates a visible exploration history and context which is built up as the programmer "expands" into the software space related to the primary source code document. Fundamentally a visible exploration context should reduce the burden on the programmer to maintain necessary context in memory and also serve as an orientation and navigation aid. Inline exploration also implicitly provides support for managing digressions. The programmer can explore a digression or side path without losing track of the original context and the digression itself is also recorded in terms of visible context. This may be sufficient for the programmer to evaluate a digression without the risk of forgetting the original goal or neglecting to return from the pursued digression.

Inline exploration also supports the simultaneous examination of related source code elements. The programmer can introduce and examine a number of source code elements into a single source code display with minimal interface adjustment. This may reduce the incidence of thrashing and support the programmer's task of understanding how program elements are related and how they interact with one another to achieve system behaviour.

4. The fluid source code editor

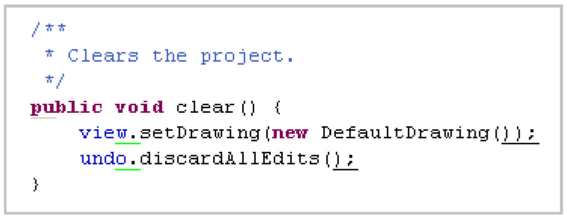
To realise the concept of inline source code exploration we developed a prototype inline exploration interface entitled the fluid source code editor. The fluid source code editor is an open source extension of the Eclipse IDE which facilitates the inline exploration of Java source code. At the core of the extension is the fluid editor, a custom Java source code editor with inline exploration capabilities.

4.1 System overview

The inline exploration support provided by the fluid editor is facilitated by two core features, visual cues embedded in the source code presentation and the inline introduction of source code declarations.

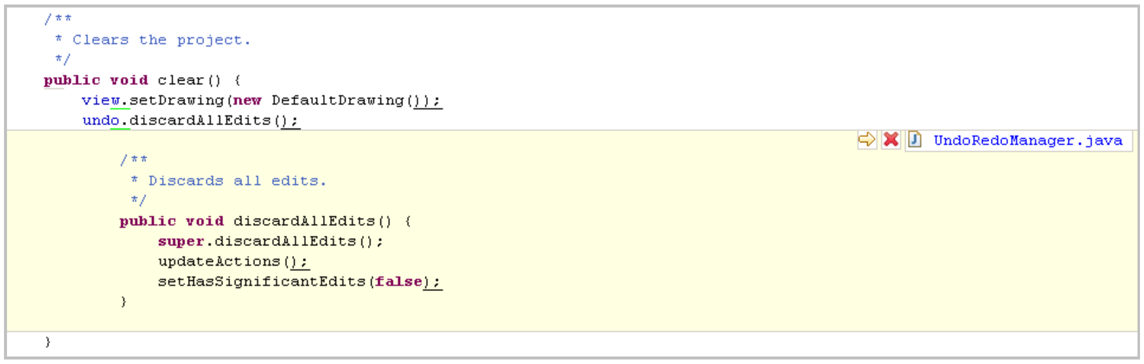
A visual cue is an unobtrusive annotation embedded in the source code editor presentation which indicates the presence of a related source code declaration to the programmer (See figure 1). When a visual cue is activated or 'opened' by the programmer the associated declaration, including any leading comments or javadoc, is dynamically introduced into the visible source code editor display (See figure 2).

Introduced source code declarations may also contain embedded visual cues which can then be interacted with to achieve nested introduction. Essentially a source code declaration may be introduced into the context of a previously introduced source code declaration (See figure 3).



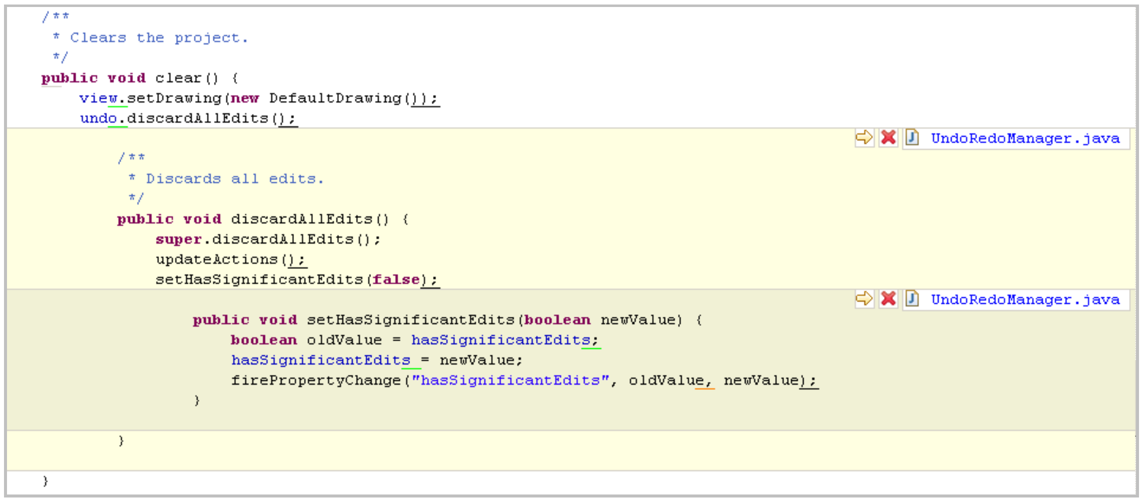
```
/**
 * Clears the project.
 */
public void clear() {
    view.setDrawing(new DefaultDrawing());
    undo.discardAllEdits();
}
```

Figure 1 Visual cues embedded in the source code presentation. Cue color indicates the type of the associated declaration.



```
/**
 * Clears the project.
 */
public void clear() {
    view.setDrawing(new DefaultDrawing());
    undo.discardAllEdits();
                                                        UndoRedoManager.java
        /**
         * Discards all edits.
         */
        public void discardAllEdits() {
            super.discardAllEdits();
            updateActions();
            setHasSignificantEdits(false);
        }
}
```

Figure 2 Introduction of an inline source code declaration, in this case a method declaration.

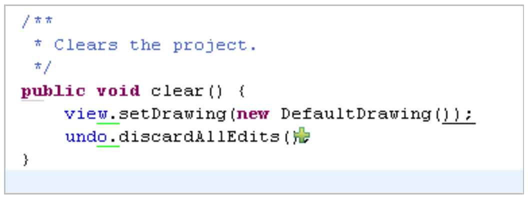


```
/**
 * Clears the project.
 */
public void clear() {
    view.setDrawing(new DefaultDrawing());
    undo.discardAllEdits();
                                                        UndoRedoManager.java
        /**
         * Discards all edits.
         */
        public void discardAllEdits() {
            super.discardAllEdits();
            updateActions();
            setHasSignificantEdits(false);
                                                        UndoRedoManager.java
                public void setHasSignificantEdits(boolean newValue) {
                    boolean oldValue = hasSignificantEdits;
                    hasSignificantEdits = newValue;
                    firePropertyChange("hasSignificantEdits", oldValue, newValue);
                }
        }
}
```

Figure 3 Nested introduction with shading applied for differentiation.

4.2 Visual cues

When a source code document is opened in a fluid editor a visual cue is associated with each cross-reference contained in the source code. The default appearance of a visual cue is a single character underline. The minimal appearance is designed to avoid eroding the readability and editability of the existing source code. When the programmer moves the mouse into the proximity of a visual cue it automatically transforms into an interactive widget, a plus affording expansion or an X affording removal or deletion of the associated inline declaration, depending on the expanded state of the cue (See figure 4). Inline introduction of the associated source code declaration is achieved when the programmer clicks on the widget associated with the visual cue.



```
/**
 * Clears the project.
 */
public void clear() {
    view.setDrawing(new DefaultDrawing());
    undo.discardAllEdits();
}
```

Figure 4 Interactive widget 'plus' associated with visual cue.

4.3 Inline Introduction

Inline introduction refers to the introduction of a related source code declaration into the context of the visible source code editor display. The fluid editor uses an inter-line introduction technique (Zellweger et al. 2000). The source code document is split horizontally at the line succeeding the visual cue and the source code declaration is then inserted as an indented code block. The expansion of multiple visual cues on a single line of source code results in the 'stacking' of inline declarations horizontally, the last introduced declaration appearing first below the anchor line.

4.4 Inline source code declarations

Inline source code declarations are read only copies of their native counterparts. The user can copy the code contained within an inline declaration but editing is explicitly prevented. While editing of the source code contained in an inline declaration is technically possible we did not pursue this feature as our focus was on exploration as opposed to out of context editing of source code. The fluid editor does however support a sophisticated editing and reconciliation system which ensures that open inline declarations remain synchronized when native declarations are edited.

Inline declarations are differentiated from native source code by a border and a coloured background. For instance, the default background color for a method declaration is a soft yellow with a grey border. In order to keep the user oriented during inline source code exploration, inline declarations are labelled with the name of their native source code document. A toolbar is also provided which allows the user to 'collapse' or delete a given declaration or navigate to the associated source code declaration in its native context. An introduced inline declaration may be 'collapsed' or removed from the source code display by re-clicking on its associated fluid annotation or clicking on the close button contained in the inline toolbar.

The fluid editor supports the inline introduction of types, methods, fields and local variables.

4.5 Nested introduction

It is common for the source code contained in an inline source code declaration to contain references to other source code declarations. To handle this scenario and fully realise the inline exploration concept, the fluid editor supports the nested introduction of source code declarations. This means that an inline declaration may be introduced into the context of an existing inline declaration.

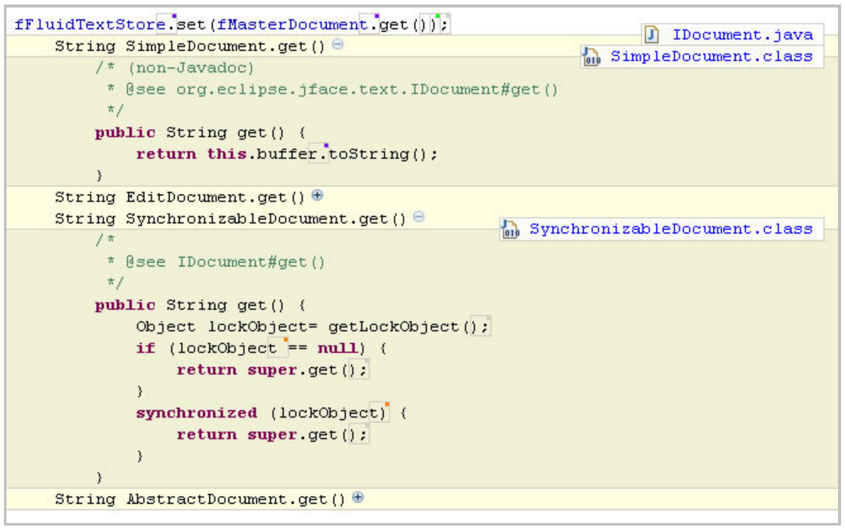
To differentiate between nested inline declarations indentation and color coding is used. A child inline declaration is indented by one tab unit greater than its parent. The fluid editor also uses shading, by default, to visually differentiate between inline declarations on different levels within an inline exploration tree. The idea is to indicate to the user that they are exploring ‘deeper’ into the software space related to the primary document. The background color of a child inline declaration is computed by taking the parent color and darkening it by a predefined factor using an RGB darkening function. When the shaded child color is deemed too ‘dark’ the procedure wraps around, the next child starts at the base default color and the process repeats. The technique assures that inline declarations which share the same background color are always distinguishable from one another and that no declaration gets shaded so dark that it would become unreadable. The fluid editor also supports an ‘alternating’ color model in which nested inline declarations are coloured from light to dark in an alternating sequence.

An entire nested inline exploration tree can be collapsed by clicking the fluid annotation widget associated with the root inline declaration. Sub-trees are also collapsible by closing the associated root inline declaration.

4.6 Search results

The core functionality of the fluid editor is the inline introduction of source code declarations. However the fluid editor framework supports the inline introduction of a number of non source code elements such as web pages, images and search results. The introduction of search results allows the user to execute searches in context. The results are displayed in place and may be further explored in an inline fashion.

Upon expansion of an interface or abstract method reference the fluid editor first runs a workspace wide search for all corresponding declarations. An inline display is then introduced containing the list of computed declarations.



```
fFluidTextStore.set(fMasterDocument.get());
    String SimpleDocument.get()
        /* (non-Javadoc)
         * @see org.eclipse.jface.text.IDocument#get()
         */
        public String get() {
            return this.buffer.toString();
        }
    String EditDocument.get()
    String SynchronizableDocument.get()
        /*
         * @see IDocument#get()
         */
        public String get() {
            Object lockObject= getLockObject();
            if (lockObject == null) {
                return super.get();
            }
            synchronized (lockObject) {
                return super.get();
            }
        }
    String AbstractDocument.get()
```

Figure 5 The inline introduction of search results, in this case the concrete implementations of an abstract method.

Each entry in the inline results list contains an associated visual cue. When the cue is expanded, the associated source code declaration is introduced. Each matching declaration can also be treated as a hyperlink facilitating explicit navigation to the associated declaration. The introduction of a polymorphic declaration takes slightly longer than a standard declaration but remains interactive. Multiple entries in the list of declarations can be simultaneously expanded for comparison and the resulting inline declarations support further nested exploration.

5. Evaluation - Design

To evaluate the effectiveness of inline source code exploration as a technique to alleviate programmer disorientation we organized a user experiment in which eight participants were asked to complete a series of exploration tasks over the source code of a moderately complex, Java based drawing application. Participants completed half of the tasks using the inline interface provided by the fluid source code editor and the remaining tasks were completed using the standard interface provided by the Eclipse IDE.

To measure disorientation we used a combination of quantitative and qualitative data gathered during the experiment. Quantitative data included task completion times, display switches and the level of interface adjustment experienced by the programmer. Qualitative data was gathered from a questionnaire associated with each interface, an exit interview and relevant comments and gestures observed during the exploration tasks. The questionnaires focused on the programmer's satisfaction with the interface as well as their feelings of disorientation and the comprehensibility of the source code explored.

5.1 Rationale

We used task completion time as a measure of performance degradation. The logic of this approach is that when the programmer experiences disorientation recovery requires additional time which then increases the overall task completion time (Edwards & Hardman 1989).

The number of display switches, or more accurately the number of times that a total replacement of visible content occurs during an exploration task is used as a measure of the 'visual momentum' of the interface. Visual momentum is 'a measure of a computer user's ability to extract relevant information across views and displays' (Woods 84). The idea is inspired from concepts used in cinematography to measure the impact from one view to another on the observer's cognitive process, in particular the extraction of task relevant information (Hochberg 1986).

When visual momentum in an information display system is low or absent, information is presented as a series of perceptually independent displays. Woods 84 explains 'Each transition to a new display becomes an act of total replacement; both display content and structures are independent of previous 'glances' into the database'. Without visual momentum between displays the user carries the burden of reorienting to the new display with each navigational transition. In contrast, an interface exhibiting a high level of visual momentum supports the continuity of structure and content from one display to another. Woods 1984 describes 'when visual momentum is high, there is an impetus or continuity across successive views that supports the rapid comprehension of data following the transition to a new display'. A high level of visual momentum between displays leads to a situation in which interface mechanisms become transparent and the user is allowed to focus fully on user level goals and tasks.

5.2 Core design

A within-subject experimental design was used where each participant acted as their own control. Participants performed a set of four tasks, one task from a set of four task types on both interfaces. The order of tasks and interfaces were systematically varied to counter-balance any skew due to practice effects. The independent variables were interface type (Standard, Inline) and task type (Local Neighbourhood, Control flow, polymorphic, inheritance).

5.3 Participants

The participants involved in the study were recruited from the computer science department at the University of Limerick. Five of the participants were graduate students one of which was also a professional programmer, two participants were faculty and one participant was a recently graduated professional programmer.

All participants were required to have strong programming experience however Java language experience and experience using the eclipse IDE varied significantly. One participant reported eight years of Java programming experience and four years using the Eclipse IDE while another reported being a novice using both Java and Eclipse. A complete list of participants and their associated level of programming and Eclipse IDE experience is presented in Table 1. The experience descriptions were transcribed from a profile questionnaire filled out by each participant at the start of their experiment session.

Although we would have liked to use only participants who were fully comfortable with both Java and Eclipse we had to compromise due to the limited availability of willing participants during the study duration.

Participant	Programming experience	Java	Eclipse
P0	4 years, C/C++, Java, Perl	Not a lot	Not a lot
P1	C, Perl, Java	Good	Experienced
P2	8 Years, Java/C++	8 Years	4 Years
P3	Java, C++, Mumps	Rusty	Very little
P4	C/C++, Java, Prolog	One year	2 months
P5	10 years, C/C++, Python	1 semester	Novice
P6	2 years C/Java, ASM	2 years	Couple of months
P7	3 Years, Java/C/C++	2 years	Intermediate

Table 1 Participant details.

5.4 Tasks

The tasks making up the study addressed both navigation and comprehension of source code during source code exploration activities. Each task was structured as a set of questions which related to particular portion of the source code. The participant was asked to read each question then explore and comprehend the associated source code and verbally or textually provide answers to the experiment facilitator. Tasks were categorized into four different types, each focusing on a particular source code exploration 'scenario'. During each experiment session the participant carried out one task of each type on both exploration interfaces.

- **Local neighbourhood**

  The local neighbourhood tasks involved the exploration and analysis of the source code 'neighbourhood' surrounding a particular source code location. The 'local neighbourhood' was defined as any piece of code which could be reached from a given source code location with three or less navigation steps.

  The local neighbourhood tasks equally emphasized both source code navigation and code comprehension. The participant was required to explore into the code space to satisfy a particular question and then backtrack to the root of the neighbourhood or a subsequent location and follow an alternate route. The participant was also required to compare related source code elements in the neighbourhood and comprehend the structure and logic of certain portions of the code spanning across source code displays.

- **Control flow**

  The second task set was described as 'Control flow'. The control flow tasks focused on the navigation and comprehension of a complex chain of control flow encompassing source code from a number of locations and documents across the code base. The average number of disjoint locations involved in a control flow was 8. The participant was guided to a particular method invocation in the code and asked to examine the structure and logic of the associated program operation driven by a number of informational goals.

  The control flow tasks emphasized the comprehension of fragmented source code and navigation through a complex control flow chain. The tasks were somewhat open ended and it was expected that the programmer would need to deal with digressions, backtracking and the perusal of alternate routes through the code. The participant was asked describe certain aspects of the operation as well as provide a high level description of the flow and functionality.

- **Polymorphic**

  The polymorphic task set focused on the exploration and comprehension of a given polymorphic operation. The participant was guided to an abstract method declaration or interface declaration and asked to answer a number of questions associated with the corresponding implementation(s) of the operation. The tasks involved comparison of concrete declarations and analysis of the type structure. The participant was also required to explore further into code space beyond concrete declarations thus performing a degree of neighbourhood exploration.

  The primary aim of the polymorphic task set was to determine how participants would react to the display of search results inline and how it would compare to the comparative functionality provided by the standard Eclipse interface.

- **Inheritance**

  The final task set focused on the exploration and comprehension of the type hierarchy associated with a given class from the point of view of the class itself. The participant was required to trace the implementation of behaviour through a number of type related method definitions, compare source code from various levels in the hierarchy, and generally examine of the structure of the hierarchy itself.

5.5 Environment

Eclipse version 3.3 was used as the overall experiment platform and provided the standard source code exploration interface. The fluid source code editor version 1.1.0 was used as the interface for the inline exploration tasks. The main Eclipse window was presented in full screen mode occupying all of the available screen space. By default the package explorer, the declaration view and the outline view were open and visible while the hierarchy view was open but stacked behind the package explorer. Participants were free to customize the eclipse window, close views and open further views as desired during the study.

Exploration tasks were based on source code associated with the JHotDraw framework version 7.0. JHotDraw (JHotDraw 2009) is an open source, Java based, 2D drawing and graphics framework including a basic drawing editor as a sample application. The JHotDraw source code is relatively small but moderately complex offering a good balance in terms of task complexity and completion times.

6. Evaluation results

6.1 Task Completion times

The completion times for all eight exploration tasks are summarized in Table 2. Overall the participants completed tasks 14 % faster on the inline exploration interface. The average task completion time on the inline interface was 588 seconds (9.8 minutes) while the average completion time on the standard interface was 679 seconds (11.9 minutes). This represents an average gain of 91 seconds (1.5) minutes on each task.

	Inline		Standard	
Task	**Mean**	**STD**	**Mean**	**STD**
Local neighbourhood A	563	283	376	62
Local neighbourhood B	513	52	614	79
Control flow A	513	184	571	240
Control flow B	446	84	576	176
Polymorphic A	481	85	554	131
Polymorphic B	553	85	621	86
Inheritance A	812	85	981	314
Inheritance B	820	216	1141	450
Average	**588**	**134**	**679**	**192**

Table 2 Task completion times.

On average, participants completed local neighbourhood exploration tasks 8% faster using the standard interface versus the inline interface. This is the only task set in which the standard interface yielded a faster completion time. The first local neighbourhood task (task A) was performed 33% faster using the standard interface however the second neighbourhood task (task B) was performed 17% faster using the inline interface. The average completion time on the standard interface was 495 seconds (8.2 minutes) and the average completion time on the inline interface was 538 seconds (8.9 minutes).

Participants performed the control flow tasks 11% faster using the inline interface. The average completion time on the inline interface was 479 seconds (7.9 minutes) and the average completion time on the standard interface was 537 seconds (8.9 minutes).

For the polymorphic task set participants performed the tasks 12% faster using the inline interface. The average completion time on the inline interface was 517 seconds (8.6 minutes) and the average completion time on the standard interface was 587 seconds (9.7 minutes).

On the inheritance task set participants performed tasks 23% faster using the inline interface. The average completion time on the inline interface was 816 seconds (13.6 minutes) while the average completion time on the standard interface was 1061 seconds (17.6 minutes).

Overall participants spent an average of 78 minutes on the inline interface and 91 minutes on the standard interface throughout the experiment session. Participants completed the exploration tasks 13 minutes faster using the inline interface.

6.2 Display switches

Overall the average number of display switches per task was 95% lower with the inline interface versus the standard interface. Using the inline interface display switches were expected to be replaced with inline introductions and thus the large reduction brought about by the inline interface was to be expected.

A more interesting look at the data is provided in Table 3 which displays the number of inline expansions and display switches on the inline interface against with the number of display switches on the standard interface.

		Inline	**Standard**
Task	**Expansions**	**Display Switch**	**Display Switch**
Local neighbourhood A	13.75	3.8	17.75
Local neighbourhood B	10.25	0	15.25
Control flow A	12.75	2	13.25
Control flow B	10.5	0.5	22.25
Polymorphic A	11	0	15
Polymorphic B	15.75	0	12.25
Inheritance A	21.75	2.3	37.75
Inheritance B	18.5	0.3	43.75
Average	**14.3**	**1.1**	**22.2**

Figure 6 Navigational actions per interface

The average number of inline introductions per task was 14.3 and the average number of display switches per task on the inline interface was 1.1. Using the standard interface participants performed an average of 22.2 display switches. As such participants performed 31% less 'navigational actions' using the inline interface.

Interestingly, on the inline interface the highest number of display switches occurred on the local neighbourhood task A. The average number of display switches was 3.8. This coincides with the fact that the participants performed local neighbourhood task A 33% faster using the standard interface. Looking at finer grain data participant p0 experienced 11 display switches when performing local neighbourhood task A and completed the task 57% slower than the average of the other three participants performing the task.

6.3 Navigational habits and interface adjustment

On the inline interface the use of the back and forward navigation actions, although supported, were negligible. Over the entire data set (32 tasks) the forward action was invoked three times and the back action was invoked 15 times. This was expected considering that the inline exploration model supports a visual exploration history which supersedes the history support provided by the IDE in most situations.

On the standard interface the back action was used, on average, 6 times per task and the forward action was used 1.6 times per task.

Horizontal scrolling was negligible and is thus not presented for consideration. Participants vertical scrolling activity increased 47% in the upward direction and 34% in the downward direction using the inline interface versus the standard interface.

6.4 Satisfaction

The results of the satisfaction questionnaires are presented in table 6. All questions were answered on a scale of zero to nine. Overall participants preferred the inline exploration interface over the standard interface, p = .0016. The inline interface scored better on the scale of terrible to wonderful, p = .018. Participants also found the inline interface easier to use p = .0012, more pleasant p = .0017, more fun p = .034 and less confusing p =.0013.

Participants agreed that they had a better idea of where they were in the code using the inline exploration interface p = .0023. There was no significant difference in how the participants perceived loss of orientation over the two interfaces p = .44. There was also no significant difference between both interfaces in terms of perceived confusion, p = .47, and wither too much information was presented on the screen at once p = .47.

In terms of the inline specific questions, participants agreed that it was easy to determine the relationship between inline source code and that visual cues were not distracting during exploration tasks. Participants also agreed that the color coding of the fluid annotations was helpful.

There was no significant difference between the two interfaces regarding the comprehensibility of the code p = .55, ability to overview the code p = .68 and ease of navigation .059. Participants agreed that it was easier to locate information in the source code using the inline interface p = .02.

	Inline		**Standard**	
Question?	**Mean**	**STD**	**Mean**	**STD**
1. How did you find the inline exploration interface in general?				
Very poor - Very good	7.7	1	5.4	1.4
2.- 6. How was the interface to use?				
Terrible - Wonderful	7.3	0.8	5.1	1.6
Hard - Easy	7.6	1	4.4	1.1
Frustrating - Pleasant	7	1	4.4	1.1
Boring - Fun	7.4	1.3	5.1	1.5
Confusing - Clear	7.3	0.5	4.4	1.5
7. It was clear, most of the time, where i was in the source code.				
I disagree - I agree	7	1.4	4.6	1.7
8. I often lost my orientation (got lost) in the source code.				
I disagree - I agree	4.7	2.7	5.7	1.7
9. I often felt confused when exploring the source code.				
I disagree - I agree	4.9	2	5.4	1.1
10. There was sometimes too much information on the screen at once.				
I disagree - I agree	5.6	2.5	4.1	2.4
11. It was easy to determine the relationships between expanded pieces of code.				
I disagree - I agree	5.7	2.4		
12. The visual cues were distracting...				
I disagree - I agree	2.3	2.7		

13. The coloring coding of the visual cues was helpful..				
I disagree - I agree	5.1	2.3	▨	▨
14. How did you perceive the tasks?				
Very poor - Very good	5.9	1.3	6.1	1.1
15. How would you rate your answers to the tasks?				
Very poor - Very good	4.7	2.4	4.9	1.6
16. - 18. Was the source code...				
Hard to understand - Easy to understand	4.6	1.7	4.1	2.1
Hard to overview - Easy to overview	4.4	2.1	4	2.6
Hard to navigate - Easy to navigate	6	2.4	4.1	1.9
19. Was information in the source code...				
Hard to locate - Easy to locate	6.3	2	4.4	2.3

Table 3 User satisfaction results.

6.5 Discussion

The results of our experiment suggest that inline exploration approach has a potential to reduce the level of disorientation experienced by programmers during source code exploration activities, and in general make source code exploration activities less cognitively demanding.

The data indicates that, on average, participants completed exploration tasks 14% faster using the inline interface and performed 31% less navigation actions. The results of the questionnaire data also indicate that participants were, overall, more satisfied with the inline interface. While none of these figures are particularly compelling on their own, together they suggest a general improvement in exploration performance.

During the study we observed participants would regularly lose context when navigating between files on the standard interface and suffer from mild disorientation before they regained context, usually via backtracking to previously visited locations. After a while participants tended to resort to using the Eclipse editor tabs to create a sort of ‘working set’ of active files which could then be indexed based on file name. In essence the participants were creating an adhoc representation of task context which was workable due to the small number of files associated with the task . Participants would then close all open editors at the start of the next task, essentially clearing the current context. We also noticed that participants would often have difficultly remembering the location of a previously visited program element or source code location and the task of re-location would disrupt the participants overall concentration on their task. We can assume that interface adjustment, getting lost and recovery of context were responsible for some of the additional time participants expended using the standard interface.

One of the participants attempted to use two editor windows in the standard interface, a feature available in eclipse allowing the user to view a number of files in a simultaneous manner. However we noticed that the participant had problems with the size of the windows being too small to view the necessary amount of source code and thus resulted in a continuous need for adjustment of window size. This participant was also observed to momentarily become disoriented when moving from the task sheet to the two editor windows, seemingly unable to remember the correct window to focus on.

Using the inline interface the participants were not observed to rely on editor tabs to maintain context and orientation. The ability to view the exploration context from the focal source code document seemed to suffice. Based on comments participants appreciated the ability to view the exploration context and compare multiple source code declarations within a single display. Participants also commented on the ability to examine a side path without moving away from the primary source code location. The ability to introduce search results inline was also a significantly commented on feature, participants mentioned that normally interface methods posed an annoyance as they had to run a search, leave the source code and continue working from the external search results view. Also once a number of results have been explored the original context of the search was lost. The average maximum number of inline declarations open at any one moment by participants was 4.

However while the usage observations and reaction to the inline interface was largely positive we did notice a number of issues. The most significant being information overload when a large number of inline declarations were introduced into a focal document. We noticed that after five or six declarations had been introduced, particularly when nested, the user would begin to become disoriented when trying to trace the relationships between inline declarations and the original context. The editor window also started to run out of horizontal space as child inline declarations would be indented further and further out. We also noticed that large declarations posed a problem in that their introduction would run off the end of the screen and the participant would need to scroll down to examine it, thus resulting in a replacement of visible context. In the scenario where participants suffered from information overload due to too many inline introduction we observed that to regain orientation participants would close all inline introductions and restart inline exploration from scratch which is undesirable. We expect that the issue of information overload could be mitigated with more sophisticated distinguishing mechanisms between nested inline declaration and/or the ability to minimize individual layers in a nested introduction tree.

Overall our experience deems that the inline exploration approach has merit, however it is also fundamentally problematic due to the limited amount of horizontal space in the focal source code document and the increasing visual complexity as a number of declarations are nested inline. The variable, and often large, size of the information being introduced also poses an issue.

6.6 Limitations

Our user experiment had a number of limitations which we shall discuss here. Possibly the most significant threat to the validity of the experiment was the design of the tasks. The eight tasks included in the experiment focused solely on the exploration of source code. However it is more realistic that programmers would both explore and edit source code simultaneously. Therefore a more realistic experiment might have participants explore source code and then make changes such as finding and fixing a bug etc. Another problem was novice effects. Disorientation may be induced by a lack of knowledge and experience with an interface such as eclipse as well as a lack of familiarity with source code (De Alwiss & Murphy 2006).

7. Conclusion

The results of our experiment suggests that inline exploration is a promising solution to programmer disorientation in certain scenarios, namely small scale exploration tasks between directly related source code locations. The results indicate that programmers spend less time and effort using the inline interface and are more satisfied with an inline interface as opposed to a standard interface alone.

7.2 Future work

A major limitation of the inline source code exploration approach is that it is only applicable to source code exploration where the programmer navigates from one source code location to a related source code location (s) via an embedded visual cue (typically a source code cross reference). However in reality programmers will navigate between source code locations, elements and documents using a variety of mechanisms, such as searching and navigating the associated search results, using the package explorer, via exploratory views such as the type hierarchy view in Eclipse and flipping between open editor tabs. Furthermore programmers will often visit a variety of other artefacts in their IDE such as xml files, web pages and so on. The inline exploration technique is unable to record and manifest this broader exploration context.

An extremely promising future area of research in terms of programmer disorientation would be to provide an IDE extension which would display a visual and interactive representation of the programmer's exploration context. For instance selecting a control key would bring up a transient semi transparent view containing a temporarily ordered graph of the current exploration history/context complete with induced digressions and the most recent elements highlighted for orientation purposes. The programmer could use this display as a reminder of context or a powerful navigation and orientation aid.

This representation could also open up interesting new areas in terms of using exploration history to tease out task context, concerns and relationships between source code elements and locations. For instance a degree of interest model (Kersten & Murphy 2005) could be applied to exploration context and the resulting model used as a summary of task context. If managed and packaged correctly a developer could load a previous developer's exploration context and use it to guide a related task. For instance a developer working on a bug associated with a particular feature could load the context recorded by the original developer as they navigated between the various program artefacts during the initial development effort.

8. References

Chu-Carrol, M.C, Wright, J., Ying, A.T.T. (2003) Visual separation of concerns through multidimensional program storage. Proceedings of the 2nd international conference on Aspect-oriented software development, 188-197.

De Alwis, B., Murphy, G.C. (2005). Remaining Oriented During Software Development Tasks: An Exploratory Field Study. Technical Report TR-2005-23, Dept. of Computer Science, University of British Columbia.

Conklin, J. (1987) Hypertext: An Introduction and Survey. Computer, 20(9), 17-41.

Desmond, M., Storey, M.A.D, Exton, C. (2006) Fluid source code views. Proceedings of the 14th IEEE International Conference on Program Comprehension. 260-263.

Eclipse.org home (2009) Retrieved May 01, 2009, from Eclipse.org: http://www.eclipse.org

Eclipse Java Development tools (JDT) (2009) Retrieved May 01, 2009, from Eclipse.org: http://www.eclipse.org/jdt/

Edwards, D.M., Hardman, L. (1999) Lost in hyperspace: cognitive mapping and navigation in a hypertext environment. Hypertext: theory into practice, 90-105.

Foss, C.L. (1989) Tools for reading and browsing hypertext. Information Processing and Management: an International Journal, 407-418.

Henderson, D.A., Card, S., (1986) Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. ACM Transactions on Graphics, 211-243.

Hochberg, J., & Gellman, L. (1977) The effect of landmark features on mental rotation times. Memory and Cognition, 5, 23–26.

Janzen D., De Volder, K. (2003) Navigating and querying code without getting lost. Proceedings of the 2nd international conference on Aspect-oriented software development, 178-187.

JHotDraw as Open-Source Project (2009) Retrieved May 01, 2009, from jhotdraw.org: http://www.jhotdraw.org

Kersten, M., Murphy, G.C. (2005) Mylar: a degree-of-interest model for IDEs. Proceedings of the 4th international conference on Aspect-oriented software development, 159-168.

Kim, H., Hirtle S.C. (1995) Spatial metaphors and disorientation in hypertext browsing. Behaviour & information technology, 14(4), 239-250.

Ko, A.J., Aung, H., Myers, B.A. (2005) Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. Proceedings of the 27th international conference on Software engineering, 126-135.

Singer, J., Lethbridge, T., Vinson, N., Anquetil, N. (1997) An examination of software engineering work practices. Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research.

Storey, M.A.D, Fracchia, F.D., Muller, H.A. (1999) Cognitive design elements to support the construction of a mental model during software exploration. Journal of Systems and Software, 171-185.

Watts-Perotti, J., Woods, D.D. (1999) How Experienced Users Avoid Getting Lost in Large Display Networks. In International Journal of Human-Computer Interaction, 11(4), 269-299.

Woods, D. D. (1984) Visual momentum: A concept to improve the cognitive coupling of person and computer. International Journal of Man–Machine Studies*, 21,* 229–244.

Woods D.D., Watts J.C. (1997) How not to have to navigate through too many displays. In: Helander M, ed. Handbook of Human-Computer Interaction, 2nd edition, 617–650.

Zellweger, P.T., Regli, S.H., Mackinlay, J.D., Chang, B.W., The impact of Fluid Documents on reading and browsing: An observational study. Proceedings of CHI'00, 249-256.