Inline source code exploration

A Thesis Submitted for the Degree of

Doctor of Philosophy

by

Michael Desmond

Department of Computer Science and Information Systems,

University of Limerick

Supervisor: Dr Chris Exton

Co-Supervisor: Dr. Margaret-Anne Storey

Submitted to the University of Limerick, June 2010.

Abstract

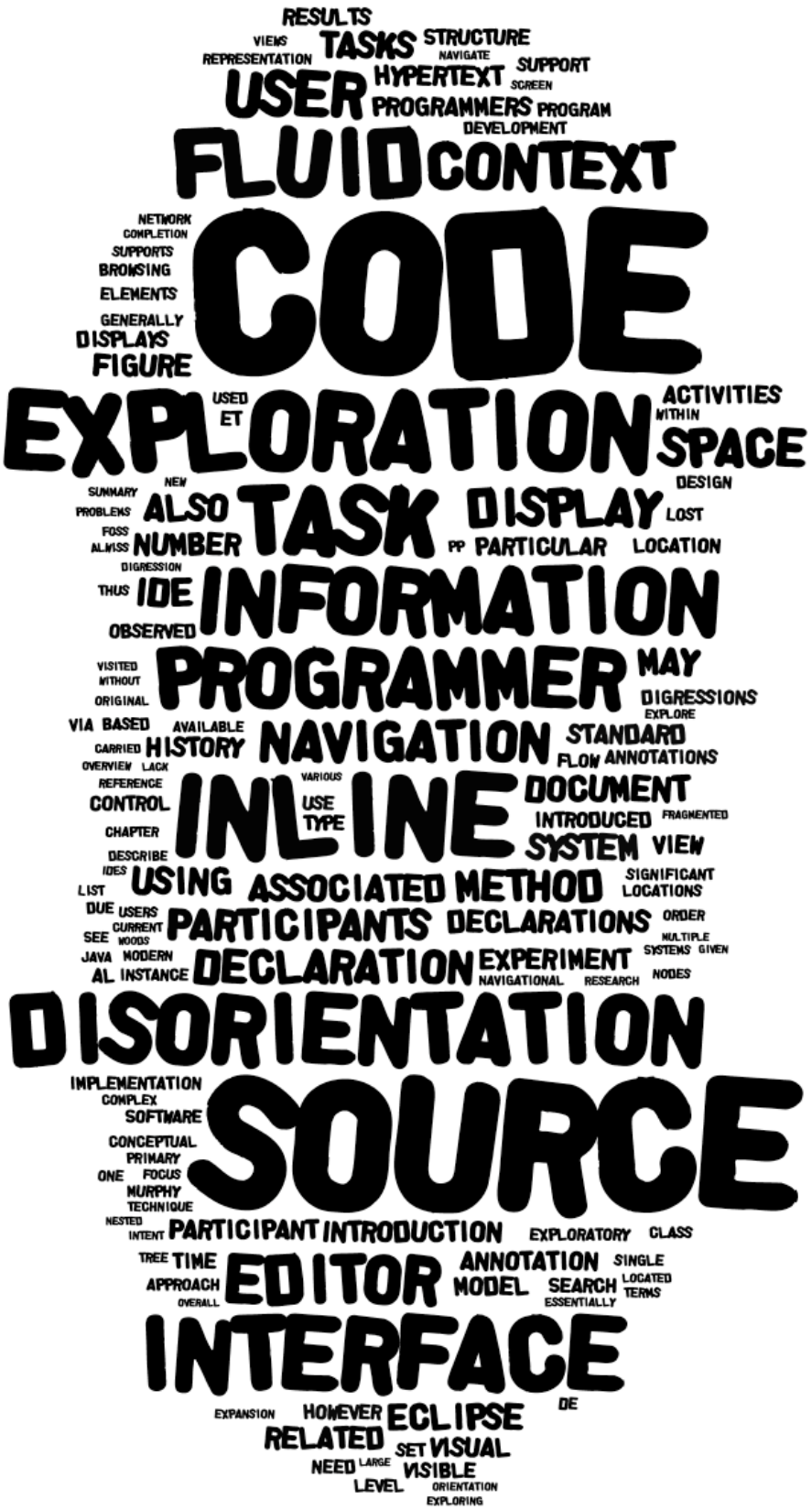
When exploring source code in modern integrated development environments (IDEs), programmers are prone to disorientation, a state of ‘mental lostness’ which disrupts concentration and task focus. Disorientation can result in important information being forgotten or overlooked and recovery requires additional time and effort which reduces programmer productivity and satisfaction.

A primary factor in the occurrence of programmer disorientation is the exploration interface design prevalent in modern IDEs. Programmers are effectively restricted to examining a single fragment of source code at any moment during exploration activities, and more significantly, there exists little or no representation or continuity of exploration history or context from one source code display to the next. Essentially, source code exploration is carried out as a series of perceptually independent glances at the code. This manner of exploration, particularly considering the complex and highly fragmented nature of source code, places a significant ongoing mental burden on the programmer and leads to a variety of problems associated with maintaining and regaining focus, finding particular locations and elements in the code and developing an accurate conceptual model of the underlying interconnected implementation.

Inline source code exploration is a mechanism for exploring source code in-context. Contrary to the traditional mechanism of explicitly navigating between isolated source code displays, the programmer progressively introduces related source code elements into the context of a focal, or primary, source code display in a controlled and

interactive manner. The inline style of exploration results in an explicit representation of the programmers exploration history/context which serves as a reminder of ongoing focus and intent as well as an orientation aid in the code space. The approach also facilitates the pursuit of exploratory digressions without the problematic need to leave the originating context and supports the examination and comprehension of fragmented source code in a single coherent display.

This thesis explores the concept of inline source code exploration and specifically its application as a means of reducing the occurrence and severity of disorientation suffered by programmers during source code exploration activities.


RESULTS
VIEWS
TASKS
STRUCTURE
NAVIGATE
REPRESENTATION
HYPERTEXT
SUPPORT
SCREEN
USER
PROGRAMMERS
PROGRAM
DEVELOPMENT
FLUID
CONTEXT
NETWORK
COMPLETION
SUPPORTS
BROWSING
ELEMENTS
GENERALLY
DISPLAYS
FIGURE
CODE
USED
ET
ACTIVITIES
WITHIN
EXPLORATION
SPACE
DESIGN
NEW
SUMMARY
PROBLEMS
ALSO
TASK
DISPLAY
LOST
FOSS
ALWSS
NUMBER
PP
PARTICULAR
LOCATION
DIGRESSION
THUS
IDE
INFORMATION
OBSERVED
VISITED
WITHOUT
ORIGINAL
PROGRAMMER
MAY
DIGRESSIONS
EXPLORE
VIA
BASED
AVAILABLE
NAVIGATION
STANDARD
CARRIED
HISTORY
FLOW
ANNOTATIONS
OVERVIEW
LACK
REFERENCE
VARIOUS
CONTROL
INLINE
USE
TYPE
DOCUMENT
INTRODUCED
FRAGMENTED
CHAPTER
SYSTEM
VIEW
DESCRIBE
IDES
LIST
USING
ASSOCIATED
METHOD
SIGNIFICANT
LOCATIONS
DUE
USERS
PARTICIPANTS
DECLARATIONS
ORDER
SEE
CURRENT
WOODS
MULTIPLE
SYSTEMS
GIVEN
JAVA
MODERN
AL
INSTANCE
DECLARATION
EXPERIMENT
NAVIGATIONAL
RESEARCH
NODES
DISORIENTATION
IMPLEMENTATION
COMPLEX
SOFTWARE
SOURCE
CONCEPTUAL
PRIMARY
ONE
FOCUS
MURPHY
TECHNIQUE
NESTED
INTENT
PARTICIPANT
INTRODUCTION
EXPLORATORY
CLASS
TREE
TIME
EDITOR
ANNOTATION
SINGLE
APPROACH
MODEL
SEARCH
LOCATED
TERMS
OVERALL
ESSENTIALLY
INTERFACE
DE
EXPANSION
HOWEVER
ECLIPSE
RELATED
SET
VISUAL
NEED
LARGE
VISIBLE
LEVEL
ORIENTATION
EXPLORING

Declaration

I hereby declare that this thesis is entirely my own work, and that it has not been submitted as an exercise for a degree at any other university.

Table of Contents

Acknowledgments

I would like to extend my sincerest gratitude to the following individuals, groups and organizations, without whose support this work would not have been possible. Dr. Chris Exton, Dr. Margaret-Anne Storey, IRCSET, all of my study participants, Harold Ossher, Jackie DeVries and IBM Research, Hawthorne NY.

- Thank you.

Chapter 1

Introduction

Chapter 1 provides a brief introduction to the thesis. This includes a discussion of core background concepts, an explanation of the problem, the proposed solution in a preliminary form and a description of the guiding research statement. The methodology, a summary of the main findings and a selection of future work is contained in the conclusion, Chapter 8.

1.1 Preamble

The study of human-centered phenomena such as disorientation, navigation and comprehension difficulties in modern integrated development environments (IDEs) is an increasingly important area of research in computer science and engineering. As software systems become ever larger and more complex programmers increasingly rely on the usability and cognitive support provided by their IDE to help them deal with the scale

and complexity of modern software systems while remaining productive and free of unnecessary stress and mental burden.

1.2 Background

Source code exploration is a fundamental and pervasive software engineering activity. Prior to, and during, software development and maintenance activities programmers will generally spend a considerable amount of their productive time exploring existing source code in order to determine and comprehend those areas or features of a system relevant to their tasks (Singer *et al.* 1997; Ko *et al.* 2005). However when exploring source code in modern integrated development environments (IDEs) programmers are prone to disorientation, a condition in which they lose the context or relevancy of their recent actions resulting in an invasive disruption to ongoing concentration and task focus (De Alwis & Murphy 2006).

Disorientation can result in relevant information being forgotten or overlooked and recovery requires additional time and effort which reduces overall programmer productivity and satisfaction.

1.3 Disorientation in the computer medium

At the root of disorientation in the computer medium is the ‘keyhole property’ introduced by Woods & Watts (1997). The keyhole property describes a common and somewhat

fundamental scenario in which a spatially large information space is examined via a small window or display capable of presenting only a very limited portion of information to the user at any given moment. In essence, the keyhole property is concerned with the spatial mismatch between a computerized information space, which is potentially vast, and the physical display space available to the user (the computer screen) which is of a limited and fixed size. The keyhole property is illustrated in Figure *1.1*.

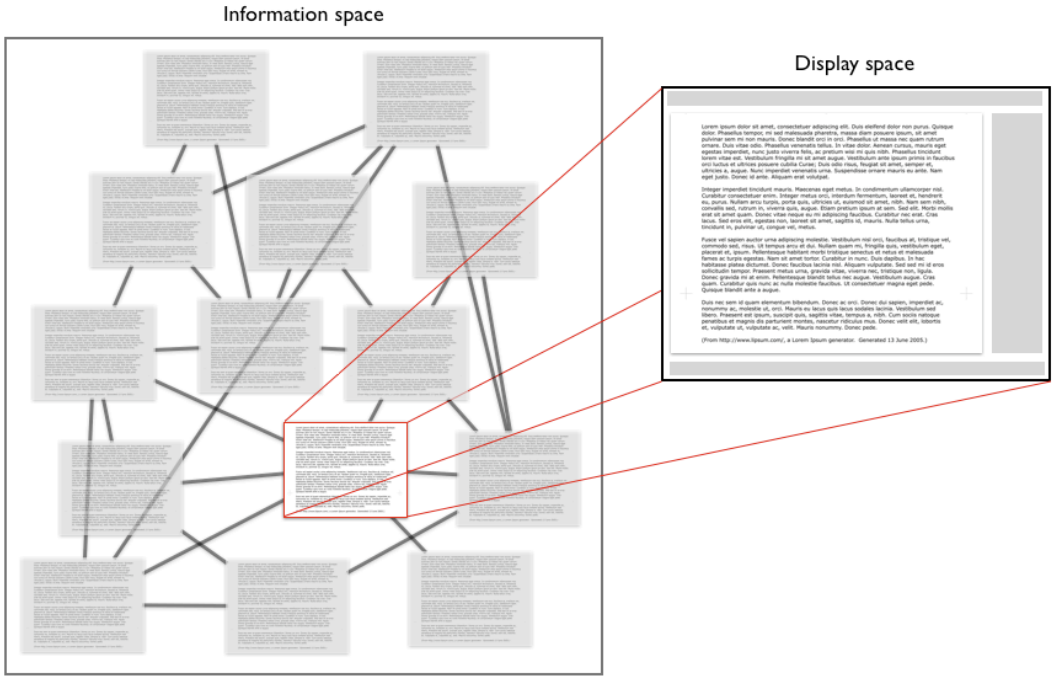


Figure 1.1: A visual representation of the keyhole property. A spatially vast information space or network is examined via a display capable of presenting only a limited amount of information to the user at any given moment. To explore the information space the user needs to 'navigate' from one location in the information space to the next examining each in isolation.

Using a keyhole interface it is generally not possible for the user to view all of the information necessary for a given task on a single display. Instead they need to call up sequences of discrete displays containing task relevant information located at various points throughout the information space. The decisions and actions which drive this process form the essence of navigation in the computer medium.

However in the absence of sufficient orientation, navigation and information synthesization support from the exploration interface the process of exploring complex information by navigating between sequences of discrete displays can result in a set of 'typical navigation problems' (Watts & Woods 1999). These problems include 'getting lost' or 'disoriented', where the user is unable to locate information, determine where they are in the information space and understand the relevance of the display they are currently examining, display thrashing (Henderson & Card 1986), where the user needs to switch repeatedly between displays in order to synthesize information from disjoint locations in the information space, and as excessive interface management, where the user needs to concentrate on extraneous interface adjustment and manipulation activities which drain attention and concentration away from primary tasks and goals (Watts & Woods 1999).

It is important to note that disorientation is a symptom of poor structure in the underlying information space in addition to the display system in use. Large, highly fragmented and irregularly inter-linked information spaces tend to be more disorientating than those which are smaller, less dense and support a more uniform and predictable structure (Maneti 1982; Van Dyke Parunak 1989). Disorientation is very much a product

of both information size and structure as well as the design of the exploration interface in use.

1.4 Programmer disorientation

A system of source code, when considered in terms of an information space, is generally quite large and exhibits a dense, complex and highly fragmented non linear structure. The widespread application of advanced decomposition technologies available in modern programming languages in addition to structural phenomena such as control flow scatter (Chu-Carroll *et al.* 2003) and the 'tyranny of the dominant decomposition' (Tarr *et al.* 1999) result in a trend whereby the implementation of conceptually coherent program operations and features is broken into a set of discrete source code fragments and elements. These units of code are then scattered over numerous disjoint source code documents and locations in the code space. Furthermore the linguistic references and semantic relationships connecting disjoint source code elements into a coherent implementation are generally dense and arbitrarily organized resulting in a complex irregular topology of information. Source code is also visually homogenous.

Reflecting the unique structure, complexity and fragmentation exhibited by source code, the act of source code exploration is characterized by repeated switching between source code documents and elements, finding and re-finding relevant information, perusal of complex branching navigation paths, and synthesization of implementation from portions of code located as disjoint points throughout code space (Singer *et al.* 1997; Ko

et al. 2005). However, modern IDEs do not provide an exploration interface with adequate support for keeping programmers oriented and focused during such activities (De Alwis & Murphy 2006).

Modern IDEs support a wide variety of exploratory navigation features and mechanisms which allow programmers to rapidly navigate through source code as they attempt to identify and piece together a mental model of system implementation. However, at any given moment the programmer is effectively restricted to examining a single fragment of source code - contained in the visible portion of the source code editor display (De Alwis & Murphy 2006). More significantly, there is little or no representation or continuity of navigation history and context or program structure from one source code display to the next. Essentially source code is explored as a sequence of perceptually independent displays, each visibly replacing its predecessor and the programmer internalizes the burden of maintaining an ongoing sense of orientation and exploration context throughout their task. This situation results in a significant cognitive overhead which is prone to breaking down due to the inherent fuzziness and limited capacity of the programmers short and medium term memory. In the event of an external interruption or when the programmer becomes momentarily preoccupied or distracted by a conceptual, navigational or interface manipulation concern, orientation and context may be forgotten or ‘pushed out’ of memory resulting in a loss of focus, and potentially a state of disorientation. To recover from disorientation a programmer may need to retrace their recent navigation actions in an attempt to rebuild context and intent or simply restart their exploration from a familiar location within the source code.

Programmer disorientation is a complex and multi-faceted problem; however, there are a number of basic factors pertaining to IDE interface design which contribute to the phenomenon (De Alwis & Murphy 2006). These factors are:

- A lack of navigation history and context
- A lack of support for pursuit of digressions
- Issues comprehending fragmented source code

1.4.1 A lack of navigation history and context

When exploring source code a programmer will generally visit and examine numerous source code locations and elements in order to identify and comprehend a particular feature or aspect of the software system. However as the programmer navigates from one discrete source code display to next there is no visible indication of how they arrived at the current source code location and its relationship to previously visited locations. This navigational context is important in terms of orientation and way-finding within the information space (Kim & Hirtle 1995) and also serves as a reminder of ongoing intent and a frame of reference in which the programmer develops their conceptual model (Storey *et al.* 1999; De Alwis & Murphy 2006).

In the absence of an explicit representation of navigation context, the programmer maintains a working sense of context in short term memory. Not only does this result in a significant ongoing mental burden, but in the event of an external, interface or conceptual distraction the programmer may forget their context and consequently their intent. Loss

of context is a common situation in modern IDEs and is often used interchangeably to describe programmer disorientation (De Alwis & Murphy 2006). Programmers will often forget how or why they arrived at a particular source code element or location and its relevance to their overall task or current exploration goals.

1.4.2 Pursuit of digressions

When exploring source code programmers are continually faced with the possibility of exploratory digressions. An exploratory digression occurs when a programmer temporarily suspends their current exploration path and intent in order to pursue (or is distracted by) a side path. Exploratory digression may in turn spawn further digressions (Embedded digressions - Foss 1989a), and eventually the primary intent may be forgotten due to an overloading of short term memory. 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 intent or simply fail to return from a pursued digression.

1.4.3 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 an overall picture or mental model of a given program feature or operation (Chu-Carrol *et al.* 2003). Because only a very limited portion of source code may be displayed at any given

moment, the programmer may need to repeatedly switch or flip between related source code displays in order gain the necessary overview or interpretative context. This behavior is known as thrashing (Henderson & Card 1986) and requires the programmer to concentrate on interface manipulation activities and maintain additional information in working memory. The distraction and interface adjustment associated with thrashing behavior may cause the programmer to lose track of their intent and become disoriented.

1.5 Inline source code exploration

To reduce the incidence of programmer disorientation and generally alleviate the mental burden on programmers during source code exploration activities, modern IDEs need to provide support for maintaining navigation history and context, managing digressions and simultaneously examining related portions of source code with minimal interface adjustment. This should then allow the programmer to focus additional mental effort on the primary task of examining and comprehending the source code and subsequently increase productivity.

A promising approach to this problem is inline source code exploration (Desmond *et al.* 2006). Inline source code exploration is a technique for exploring source code *in context*, meaning that fragments of source code may be examined in parallel with surrounding or adjacent source code during exploration tasks and activities. The essential premise is that instead of explicitly navigating between isolated displays of source code (See Figure *1.2*), which results in a continuous loss of navigation context, the

programmer progressively introduces related portions of source code, inline, into the context of a focal source code editor display (See Figure *1.3*).

Figure 1.2: Explicit navigation between discrete source code displays. Each new display replaces the previous during the exploration process. This style of exploration results in a loss of navigation context, problems associated with exploratory digressions and difficulty comprehending fragmented source code.

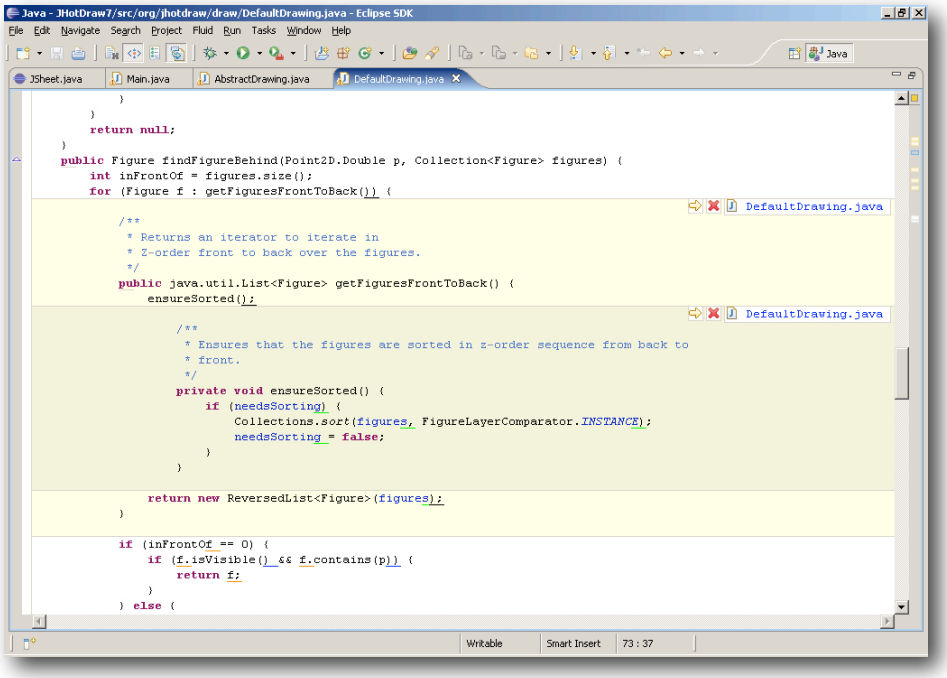


Figure 1.3: Inline exploration. Each new source code element is introduced into the

context of the focal source code display. A representation of the programmers navigation history is maintained in terms of the visited source code elements themselves.

Inline exploration of source code facilitates a visible navigation history and context which is built up as the programmer progressively "expands" into the software space related to the primary source code document. Fundamentally a visible context should reduce the burden on the programmer to maintain necessary navigational context in memory and also serve as an orientation and navigation cue and a visible reminder of exploratory intent.

Inline exploration also implicitly provides support for managing digressions. The programmer can explore an exploratory digression or side path without losing track of the original context and the digression itself is also recorded in terms of visible navigation context. This may be sufficient for the programmer to evaluate the 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 multiple related source code elements in a single display. 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 thus support the development of an accurate conceptual model.

1.6 Research

The goal of this research is to explore the use of inline source code exploration as a technique to alleviate programmer disorientation, and reduce the general mental burden on programmers during source code exploration activities. The work aims to determine the feasibly of an inline source code exploration model, identify the advantages and disadvantages of the approach, and explore design concerns and usability. The guiding research questions is thus:

What are the advantages and disadvantages of inline source code exploration, and what effect does it have on programmer disorientation?

Chapter 2

Disorientation in the Computer Medium

"*-the user does not have a clear conception of relationships within the system, does not know his present location in the system relative to the display structure, and finds it difficult to decide where to look next within the system.*"

- Elm & Woods (1985).

Disorientation has been identified and studied in a variety of domains such as hypertext systems (Conklin 1987; Foss 1989a; Foss 1989b; Kim & Hirtle 1995; Edwards & Hardman 1999), spread sheets (Watts-Perotti & Woods 1999), menu systems (Maneti 1982), software development environments (De Volder 2003; De Alwis & Murphy 2006; Kersten & Murphy 2005), computerized nuclear power plant operating instructions (Woods & Roth 1988) and medical monitoring systems (Cook and Woods 1996). Disorientation refers to a temporary state of mental lostness which disrupt a users

concentration and focus during complex tasks in the computer domain. The phenomenon can result in important or relevant information being forgotten or overlooked, and recovery requires additional time and effort which negatively affects a users productivity and satisfaction.

This chapter describes what disorientation is, and why it occurs in the computer medium. A particular focus is on the type(s) of disorientation which occur during hypertext exploration activities. The justification for this focus is that many of the disorientation problems experienced by programmers during source code exploration are very similar to those experienced by users of hypertext systems (De Alwis & Murphy 2006; Storey *et al.* 1999). This reflects the very similar structure shared by both information mediums. However the investigation and understanding of disorientation in the hypertext domain is significantly more mature and thus presents a convenient platform on which to understand the phenomenon at a level of depth which is currently unavailable in the source code exploration literature.

2.1 Fundamentals

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 available physical display space available to the user - represented by the physical

computer screen. This mismatch is referred to as the “keyhole property” (Watts & Woods 1997) and illustrated in Figure *2.1*.

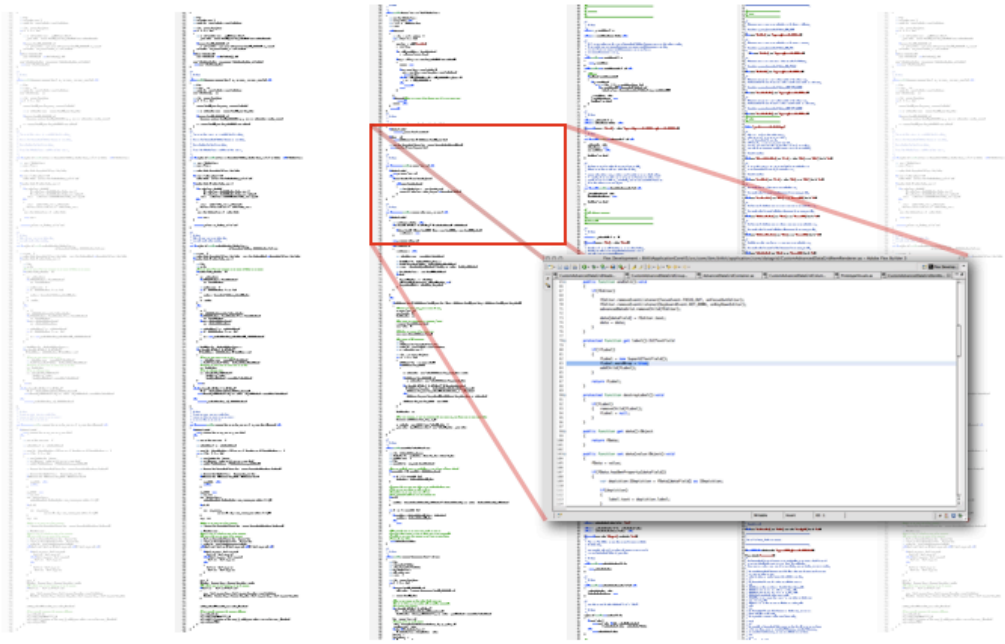


Figure 2.1: The keyhole property. A spatially large information space is examined via a limited display.

Examining a large information space via a keyhole display means that it is generally not feasible for the user to examine all of the information required for a given task simultaneously in a single display. Instead the user needs to decide which portions 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 the keyhole display makes it difficult for the user to locate and remember information and to synthesize a coherent overall picture from pieces of information located at disjoint locations throughout the information space. Typical

problems that users experience are ‘getting lost’ or ‘disoriented’ in the information 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 & Woods 1997).

An example of the problems associated with a keyhole display is reported by Elm & Woods (1985). Designers attempted to convert the existing paper based instructions for nuclear power plant emergencies procedures into a computerized format similar to html pages.

people became disoriented or “lost,” unable to keep procedure steps in pace with plant behavior, unable to determine where they were…(in the display space), unable to decide where to go next, or unable even to find places where they knew they should be (i.e., they diagnosed the situation,knew the appropriate responses as trained operators yet could not find the relevant procedural steps in the network).

- Woods & Roth (1988) p. 10

It is telling to note that the operators became so disorientated by the computerized instruction system that they had to abandon their task completely. This is the essence of disorientation in the computer medium.

Two other factors are also of fundamental importance when considering disorientation in the computer domain, the structure of the underlying information space and the structure of display interface in use. Large, irregular information spaces tend to be more disorientating than those which are small and more uniform. The introduction of some semblance of uniformity within an information network can dramatically improve navigability and aid in preventing disorientation (Van Dyke Parunak 1989). However regardless of the size or uniformity of the underlying information space, It is the role of the display interface to expose the underlying structure and help the user navigate in the space without becoming disorientated.

2.2 Lost in hyperspace

A classic example of a keyhole scenario is represented by hypertext systems. A hypertext system is composed of a database containing textual nodes of information linked together into an information network (a hyperspace) and a display system capable of presenting the contents of a node in an on screen display. A node display may contain any number of links, presented as selectable portions of text or other visual artifacts, which represent pointers to other nodes in the information network. When a link is selected by the user the referenced node is located and opened in the on-screen display. A prototypical hypertext system is illustrated in Figure *2.2*.

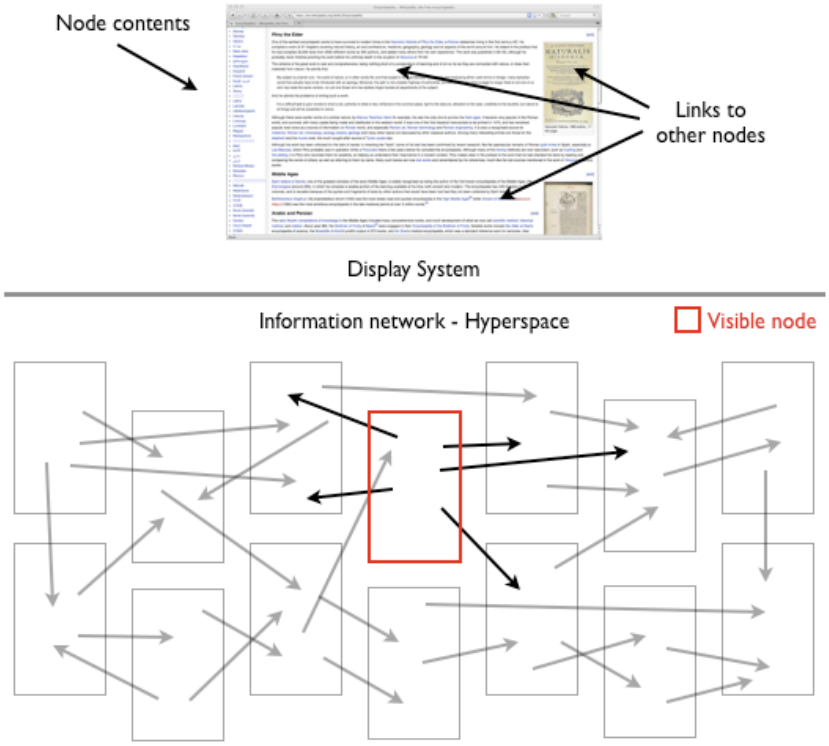


Figure 2.2: A hypertext system which is composed of an underlying network of information nodes and directed links (the hyperspace), and a display system capable of displaying the contents of a node on screen and facilitating navigation between individual nodes via links.

Browsing or exploring a hypertext system refers to the process of visiting sequences of nodes (displays) and links to locate and assimilate information related to user tasks and goals.

During hypertext browsing activities three distinct categories of disorientation have been identified in the literature (Foss 1989a; Foss 1989b). These categories are:

- Navigational disorientation
- Task disorientation
- Informational disorientation

2.2.1 Navigational disorientation

Navigational disorientation also known simply as ‘disorientation’, ‘getting lost’ (Conklin 1987) or ‘being lost in hyperspace’ (Edwards & Hardman 1989) occurs when a user becomes spatially lost when exploring a hypertext system. Conklin (1987) describes the problem as ‘the tendency to lose ones sense of location or direction in a non linear document’.

A hypertext system facilitates the fragmentation of a coherent body of information into a collection of textual nodes connected via a network of directed links. When exploring a hypertext system the user needs to maintain a sense of where they are in the network, how they arrived at the current location and how to navigate to nodes which are known or expected to exist (Thüring *et al*. 1995). This information is important for way-finding and navigation activities (Kim & Hirtle 1995) and understanding the semantic content of the system and developing an accurate conceptual model (Thüring *et al*. 1995).

In the absence of external orientation aids provided by the display interface the user maintains a sense of orientation in working memory which tends to be both limited and erroneous. As a result, in large topologically complex information networks users are prone to losing track of their location and having difficulty remembering and finding information of interest. Foss (1989a) explains that navigational disorientation is an inherent aspect of hypertext systems as a hyperspace may contain hundreds of nodes of information linked together in a complex topology yet only a very small number may be examined on screen at any particular moment.

Symptoms of navigational disorientation include failure to locate information in the network (retrieval failures), navigational problems such as looping and inefficient paths, lack of closure (uncertainty whether the extent of the network is known or whether information may have been forgotten or passed by) and not knowing if a link will bring up relevant or sought after information (Foss 1989b). Edwards and Hardman (1999) characterize navigational disorientation as:

- Not knowing where to go next
- Knowing where to go, but not knowing how to get there
- Not knowing the current position relative to the overall structure

To recover from navigational disorientation users have been observed to return to a familiar location within the information network to restart exploration or backtrack to previously visited nodes in an attempt to regain context (Foss 1989b).

2.2.2 Task disorientation

Task disorientation also known as ‘task overload’ (Foss 1989a) or ‘cognitive overhead’ (Conklin 1987) refers to problems managing and keeping track of goals and digressions during exploration activities. These problems originate from the cognitive demands placed on the user when exploring a complex information system. Conklin (1987) describes cognitive overhead as “the additional effort and concentration necessary to maintain several tasks or trails at any one time”. The phenomenon is succinctly described by Foss (1989a) in terms of the “embedded digression problem”.

The embedded digression problem describes the process by which a user suffers from disorientation when pursuing multiple paths or digressions through an information network. When following a path through the network, which is generally driven by a particular task or goal, the user may encounter an interesting aside thus branching from the main exploration path into a digression. The current task is mentally put ‘on hold’ by the user as the digression is processed. However the digression may take an unexpected amount of time, or may in turn spawn further digressions (embedded digressions) which can lead to the original task(s) being forgotten due to the limited attention and working memory of the user. McAleese (1999) elegantly summarizes the phenomenon as when “the user keeps following chains of thought until the original goal is lost”.

It is interesting to note that exploratory goals may also be forgotten when users are required to concentrate on navigational or interface management issues such as locating nodes in the network, recovering from problems such as getting lost and arranging the interface (Foss 1989a).

Symptoms of task disorientation include users forgetting why they moved to a particular location (What was I doing?), confusion as to what they had intended once arriving at a location (What was I going to do?), forgetting to return from digressions and neglecting to pursue planned digressions (Foss 1989b).

2.2.3 Informational disorientation

The final category of disorientation experienced during hypertext exploration activities is informational or conceptual disorientation. Informational disorientation pertains to difficulties comprehending, synthesizing and summarizing the semantic content of the information examined during a period of exploration. Foss (1989a) describes this problem in terms of the "Art museum phenomenon". The 'Art museum phenomenon' describes a situation where subsequent to viewing a large number of nodes of information the task of forming a coherent abstraction or comprehension of the overall content can be difficult for the user. The art museum metaphor is derived from a hypothetical visit to an art museum in which a person might look at hundreds of paintings yet be unable to describe a particular painting in detail or describe how particular styles of art have influenced one other.

Symptoms of the art museum phenomenon include a lack of detailed memory for any particular node that was examined and an inability to summarize or abstract what was learned during a exploration or browsing session (Foss 1989a).

2.3 Factors inducive to disorientation

Disorientation experienced during information exploration activities is a factor of both the structure of the underlying information network as well as the display system or exploration interface in use.

2.3.1 Structure of the information space and its role in disorientation

Van Dyke Parunak (1989) explains that the complexity of the topology connecting an information network affects a user's ability to navigate within that space. Simple, regular topologies, such as linear, ring or hierarchy allow the user to employ a variety of navigation strategies and the scope for users getting lost and disoriented is low. There is a definite spatial organization within the network which the user can exploit as an orientation and navigational aid. For instance Conklin (1987) notes that in a linear network the user has two choices when attempting to find information, either before the current location or after. In an arbitrarily complex network the choice is not so straight-forward. Van Dyke Parunak (1989) explains that as the size and topological complexity of the network increases, the number of available navigation strategies decreases and there exists more ways for the user to move between different nodes of information. This richness and variety of navigational options leads to problems such as getting lost and disoriented. Van Dyke Parunak (1989) concludes that arbitrary topologies offer the least number of navigation strategies and thus the greatest scope for user disorientation.

Kim & Hirtle (1995) note that path complexity is a factor which negatively affects navigation performance and the acquisition of mental maps in a hypertext system. Path complexity refers to the number of decision or branch points along paths through the network. Foss (1988) notes that when a lot of interesting links are located in close proximity the user may be distracted from their main task and suffer from embedded digression problems leading to task disorientation.

Thüring *et al.* (1995) describe that 'the fragmentation characteristic' inherent in hypertext systems hinders a user's ability to construct a mental representation of information distributed across a number of nodes in a hyperspace. Essentially the greater the level of fragmentation in a hyperspace (the finer the granularity of related information nodes) the more effort required by the user to synthesize an overall mental model due to the keyhole effect (only a limited number of nodes may be displayed on the screen at any given moment).

Differentiation and homogeneity are also significant factors in terms of potential for disorientation in an information space. Differentiation refers to the ability to distinguish between fragments of information (Kim & Hirtle 1995) and homogeneity (Nielsen 1990) refers to a situation where the appearance or presentation or information in a information space makes it difficult for users to differentiate. When the information contained in an information space is visually homogenous users are unable to recognize previously visited locations, thus making them confused about the path they have taken (Nielsen 1990).

In summary large, irregular, highly fragmented and visually homogenous information networks are much likelier to induce disorientation on behalf of users.

2.3.2 Structure of the exploration interface and its role in disorientation

While poor structure of the underlying information network is a factor inducive to disorientation, it is the role of the display interface to help the user maintain orientation, synthesize information across displays and avoid navigation problems.

Watts-Perotti & Woods (1999) maintain that the degree of visual momentum exhibited by a display interface dictates the level of disorientation and navigational problems experienced by the user.

Visual momentum is ‘a measure of a computer user’s ability to extract relevant information across views and displays’ (Woods 1984). 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 discrete isolated displays. Woods (1984) 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 mental burden of maintaining exploration context and reorienting to the new display with each navigational transition.

In contrast, an interface exhibiting a high degree 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.

Essentially high visual momentum implies that a user expends little mental effort to place new displays in the context of the larger system. Conversely, low visual momentum requires users to spend more mental effort in putting new displays into the existing context. Design techniques to increase visual momentum include display overlap, long-shots, landmarks, bookmarks and spatial dedication (Watts-Perotti & Woods 1999).

2.3.2.1 Long-shots

A long shot is a high-level overview of the information space which the user can exploit as an orientation, navigation and comprehension aid. The long shot allows the user to relate the current display to previous displays and to establish a frame of reference to aid in comprehending upcoming displays, thus helping to alleviate both navigational and task disorientation.

An example of a long shot is a graphical overview of the underlying information network which highlights the users current location along with previously visited and upcoming locations.

2.3.2.2 Functional-overlays

A functional overlay displays the context surrounding the display currently being examined. Generally this allows the user to avoid navigation and consider multiple related pieces of information in a single display, thus helping to relieve navigational, task and informational disorientation.

An example of a functional overlay is a pop-up window containing the contents of a link in a hypertext system. The user can examine the contents of the pop-up window while simultaneously examining the original link location and its surrounding context.

2.3.2.3 Landmarks

A landmark is a prominent and easily identifiable feature or element in the information space which, at a glance, provides information about orientation and location to the user. Landmarks are used to aid users in reorienting to the structure of the information space as well as providing a mechanism to return to known locations. The classic example of a landmark is the ‘home’ button in a hypertext system.

2.3.2.4 Bookmarks

Bookmarks record positions of interest in an information space. The use of bookmarks allows the user to store a set of pertinent locations in a central and easy to access area of the display space. This means that the user can access relevant information without the need to exhaustively search through the network. Removing the need to exhaustively search alleviates navigational and task disorientation. Furthermore bookmarks may be used as a record of the interesting elements discovered during a complex task thus helping to alleviate informational disorientation.

2.3.2.5 Spatial dedication

Spatial dedication is concerned with the organization of the information space so that information is spatially arranged. This means that information is consistently kept in a certain place within the system. Spatial dedication provides users who are familiar with the spatial structure a memory aid facilitating navigation to desirable information without the need to exhaustively search through the information network. This alleviates both navigational and task disorientation.

Watts-Perotti & Woods (1999) describe spatial dedication as a mechanism used to remain oriented in spread sheet environments. Users were observed returning to known locations in a given sheet based on natural spatial encoding. This mechanism allows the user to remain focused on their task. De Alwis & Murphy (2006) also observed programmers using specific spatial locations within source code documents to quickly

locate frequently visited method bodies and the location of variable definitions (generally at the top of a given source file) without exhaustive scrolling.

2.4 Tools and techniques to alleviate disorientation

In response to the various categories of disorientation encountered when exploring hypertext systems a variety of technologies have been developed to alleviate the various aspects of the phenomenon and support the user to remain focused and oriented in the information space. Many of these techniques are based on or match the the design techniques to increase visual momentum as outline in Chapter 2 Section 3.

2.4.1 Overview maps

Conklin (1987) proposes that one of the major technological solutions to the problem of navigational disorientation is the use of graphical browsers or overview maps. A graphical browser is a tool capable of rendering the structure of the underlying information network in a 2d or 3d display thus creating a virtual spatial environment which can be exploited by the human visiospatial system. Typical examples of graphical browsers appear in hypertext systems such as NoteCards (Halasz *et al.* 1986) and Intermedia (Garrett *et al.* 1986). Watts-Perotti & Woods (1999), in a study of spreadsheet users, found that the provision of a printed map of a spread sheet structure reduced perceived disorientation on behalf of users.

Thüring *et al.* (1995) also propose that to decrease the cognitive overhead on users, and thus increase the comprehensibility of a hypertext system, a graphical map or presentation of the information space should be provided which enables the user:

- To identify their current position with respect to the overall structure
- To reconstruct the way that led to the current position
- To distinguish between different options for moving on from this position

However there are a number of challenges with respect to overview maps which limit their usefulness at alleviating disorientation. Due to screen space limitations only a small portion of a very large network overview may be visible to the user at any particular moment. Furthermore as the complexity and density of the network increases the overview becomes more difficult to use and interpret and thus its usefulness quickly dissipates. Foss (1989a) explains that "browser graphs containing more than 10-15 nodes approached the user's saturation point". In such a scenario a form of distorted view or fisheye is generally required - or alternatively the overview may be filtered to display only pertinent nodes based on some defined criteria.

2.4.2 Search/Query mechanisms

A second technological solution to navigational disorientation proposed by Conklin (1987) is the application of database style search/query mechanisms. Instead of browsing for nodes/links in the network, the user executes a search based on criteria such as

keywords, arbitrary strings or node/link attributes such as author, type etc. The user can then navigate directly to the required information and bypass the need for searching through the information network.

Fast arbitrary navigation systems are also important in terms of task disorientation. For instance a user might be exploring at one point in the network and suddenly want to check some information located at a distant node. By the time the user has navigated across the network, dealing with the associated navigational overhead, they may have forgotten their original intent or become lost.

2.4.3 Guided Tours/Beaten tracks

A guided tour or beaten path (Van Dyke Parunak 1989) is a predefined or recommended path through an information network which is laid out by the author or a power user. The user relinquishes the flexibility of open exploration by following a series of clearly marked links and nodes. The task and navigational disorientation associated with browsing is avoided and the user is free to concentrate on reading and comprehending the information content. However guided tours are still prone to conceptual disorientation associated with information fragmentation. Guided tours are considered useful for introducing a hyperspace to new or novice users.

2.4.4 Visual/Interactive browsing summary/history

Foss 89a proposes an innovative suite of hypertext browsing tools designed to alleviate the disorientation associated with embedded digressions and the art museum phenomena. The primary facet of the tool suite is to provide the user with a visible and interactive history of their browsing activities.

2.4.4.1 Graphical history list

A graphical history list (Foss 1989a) is designed to support the management of exploratory digressions and promote user orientation by providing a list of the nodes and local neighbourhoods visited by the user during a browsing session (See Figure *2.3*). The list is displayed on screen and contains an entry for each node visited during a session. When an entry is selected a 'mini browser' is opened showing the local neighbourhood associated with the visited node. The mini browser distinguishes between visited and non visited neighbouring nodes and may be computed to a given depth.

The graphical history list supports digressions as users have a record of the nodes that have been visited and can easily return to unexplored neighbouring nodes via the mini browsers. The mini browser also allows the user to regain context at previously visited nodes and distinguish unexplored neighbouring nodes.

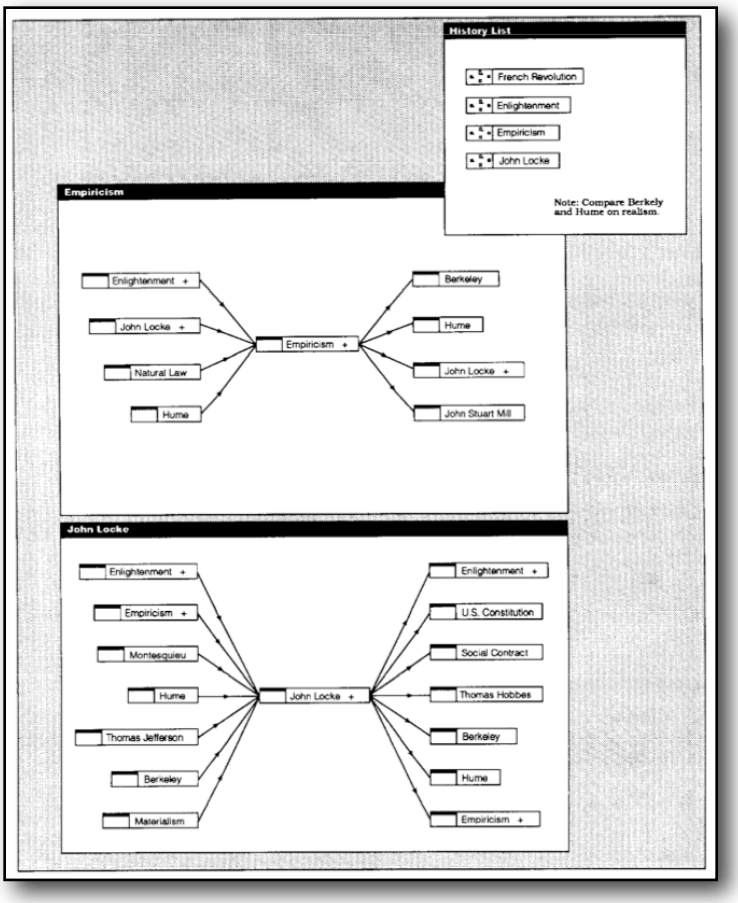


Figure 2.3: A graphical history list (top right) containing an ordered list of the nodes previously visited by the user. 'Mini-browsers' depicting the local neighbourhood surrounding the selected nodes are displayed in centre.

2.4.4.2 Graphical history tree

A history tree (Foss 1989a) is similar to a graphical history list in that it records the nodes visited by a user during a browsing session. However the history tree displays the user's navigation path hierarchically and includes a precise temporal ordering (See Figure *2.4*).

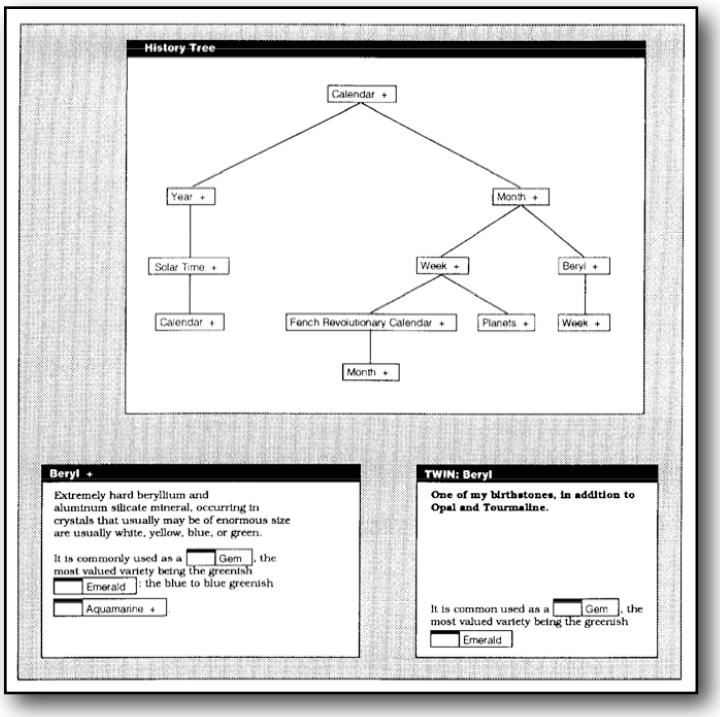


Figure 2.4: A graphical history tree displaying the users navigation path through the hyperspace. Branches indicate digressions inferred from cycles in the users navigation history.

When a cycle is detected in the navigation path it is treated as a completed digression and added to the tree as a branch with the revisited node as root. The history tree allows the user to visualize their navigation path through a hypertext system in terms of paths and digressions. Selecting an entry in the history tree opens the corresponding hypertext node in an onscreen display.

2.4.4.3 Summary boxes

Summary boxes (Foss 1989a) are designed to facilitate note taking during a hypertext browsing session, thus aiding informational tasks. If a summary box is open on screen all nodes visited by the user are opened with a blank 'twin' node. The user can enter notes in the twin or copy interesting information from the original node into the twin. The original node and the twin are dynamically entered into a history list contained in the summary box (See Figure *2.5*).

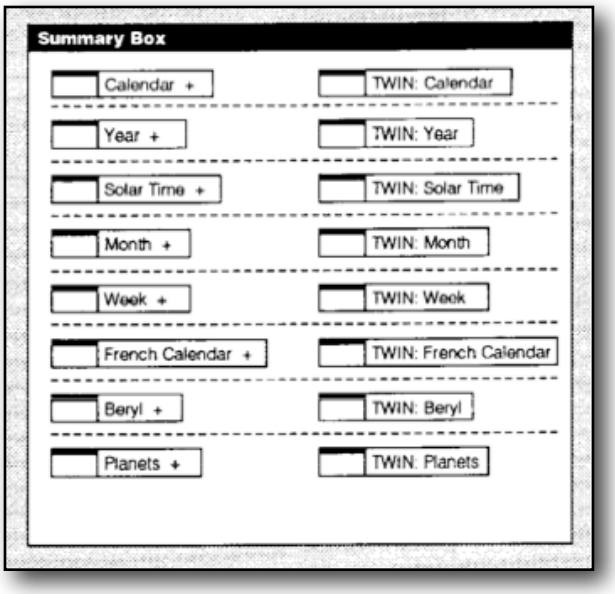


Figure 2.5: A summary box containing the visited nodes and associated 'twin' or summary cards. Pertinent notes and information may be copied from the original nodes into the summary cards during browsing activities.

The summary box is designed to mitigate the art museum phenomenon as the user may examine the summary box after a browsing session and view a list of the frames visited annotated with notes and other information deemed pertinent during the session.

2.4.4.4 Summary tree

Summary trees (Foss 1989a) are much like history trees in that they record the user's movements through the hypertext network as a temporally ordered hierarchy of nodes. However in a summary tree the user is free to add annotations to the tree in the form of notes and arbitrary links between nodes. An example summary tree is illustrated in Figure *2.6*. The summary tree is designed to allow the user to build a custom conceptual map or diagram which may or may not be isomorphic to the actual structure of the hypertext network itself. The conceptual diagram helps the user to summarize and comprehend the information encountered during a browsing session.

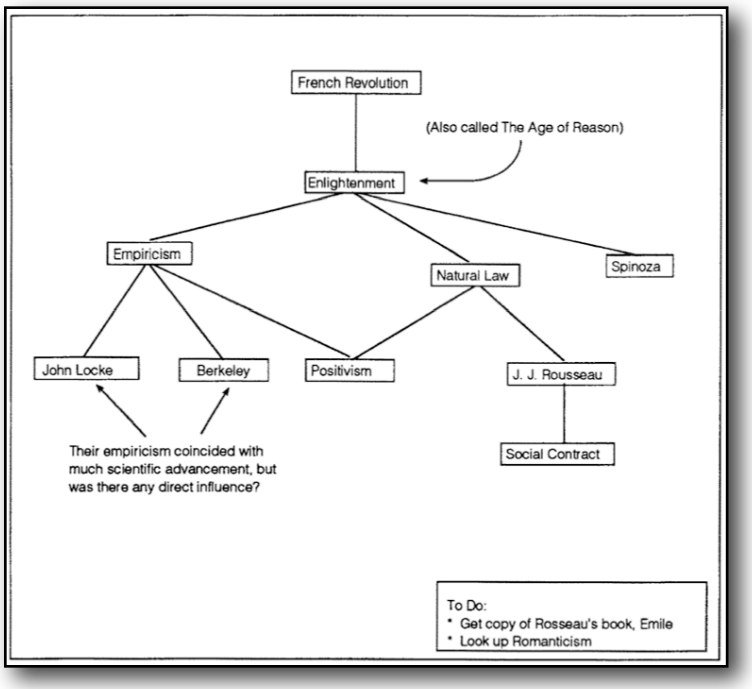


Figure 2.6: An annotated summary tree. The summary tree allows the user to arbitrarily annotate a representation of their browsing history.

2.4.5 Preservation of context

Preservation of context is a technique which allows the user to examine information in the context of previously examined information. The SEPIA browsing interface (Thüring *et al.* 1995) displays the contents of the current or active information node and the contents of its immediate predecessor in a dual display as the user explores the system. Thüring *et al.* (1995) explain that preserving the context of the previously explored node supports the "given-new-strategy" (Clark and Haviland 1974). When new information is displayed in context with previous information it becomes easier for the user to make meaningful connections and relationships between the information units. The impression of fragmentation in the information network is also reduced.

2.5 Summary

This chapter discussed the core tenets of disorientation in the computer medium, with a particular focus on the hypertext exploration domain. A particular focus on hypertext was made as it shares a very similar structure to source code.

Disorientation refers to a state of mental lostness which users encounter when exploring or browsing large information spaces. During hypertext browsing activities three types of disorientation have been identified, navigational disorientation where the user loses their sense of location and orientation in the information network, task disorientation where the user has difficulties managing tasks or digressions and informational disorientation where the user has difficulties comprehending and summarizing information.

Disorientation is induced by the structure of the underlying information network as well as the structure of the display system in use. Large, fragmented and visually homogenous information spaces exhibiting an irregular topology have the greatest scope for user disorientation. A high level of visual momentum in the display interface reduces the incidence of disorientation and helps users focus on tasks and goals.

A variety of technologies have been developed to alleviate the various aspects of disorientation during hypertext exploration activities. The technologies include overview maps, search/Query mechanisms, the provision of interactive browsing history and preservation of context.

Chapter 3

Lost in Code Space

"*During informal conversations, developers have mentioned to us that they occasionally become lost or disoriented when they explore a system. The disorientation involves losing the context or relevancy of their recent actions to their overall goal.*"

- De Alwis & Murphy (2006).

Programmers have reported, and have been observed to suffer from disorientation during source code exploration activities, even when working in advanced modern integrated development environments (De Alwis & Murphy 2006). This chapter discusses programmer disorientation, specifically in the context of source code exploration activities in an IDE setting.

Initially a number of structural properties of source code which are inducive to disorientation are discussed. This perspective provides an important foundation for ongoing discussion and links into the discussion of information structure and its role in

disorientation discussed in Chapter 2 Section 3.1. Then an analysis of the interface design of a state of the art modern IDE is presented. The focus of this analysis is to identify interface characteristics and limitations which contribute to the disorientation, such as lack of visual momentum (Chapter 2 Section 3.2). Finally a number of existing technologies and research tools designed to alleviate disorientation in the IDE are described and their scope and effectiveness considered.

The aim of this chapter is to explore the conceptual landscape associated with programmer disorientation in modern IDEs and highlight issues in the space.

3.1 Source code exploration

Before discussing the issue of programmer disorientation during source code exploration activities, it is useful as a preamble to first consider the nature of source code exploration itself. Contrary to the myth of a new programmer joining a development team and attempting to 'learn the system' in a single exploratory session, source code exploration is generally carried out on an 'as needed' basis and in the context of specific software development and maintenance tasks (Singer *et al.* 1997).

Due to the sheer size and complexity of modern software systems it is generally not feasible for an individual programmer to maintain a detailed knowledge or comprehension of a software system on an ongoing basis. Instead programmers will explore and comprehend portions of the system as needed, driven by their immediate task requirements. This generally involves exploring just enough of the system to identify

those constructs and interactions implementing a particular aspect or feature prior to making necessary changes or enhancements.

Once a change task has been satisfactorily completed, the in depth-knowledge of the source code gained during the exploration and discovery phase rapidly fades into an abstract overview. The programmer will simply re-explore portions of the system in detail when encountered during subsequent tasks.

This high level approach to exploration and discovery in software systems has been described as ‘just in time comprehension’ (JITC) (Singer *et al.* 1997) and highlights the core and pervasive nature of source code exploration as a programmer activity.

3.2 Disorientation in code space

Programming languages provide linguistic tools and constructs which facilitate the decomposition of complex systems and operations into discrete program elements (for instance interface, type, method and fields declarations in an object oriented system). These elements are organized into a set of physical storage units - generally source code documents - and linked using a variety of expressive referencing mechanisms to achieve desired system structure and behavior. In essence, source code may be considered as a non-linear information space composed of inter-referenced program elements, similar in structure to a complex hyperspace (Storey *et al.* 1999; Schümmer 2001). See Chapter 2, Section 2 for a discussion of hypertext systems and the structure of a hyperspace for reference.

Programmers repeatedly explore within ‘code space’ in order to assimilate a comprehension or ‘mental model’ of system implementation prior to and during software development and maintenance activities (Singer *et al.* 1997;Ko *et al.* 2005). This conceptual model allows programmers to reason about the system and make informed decisions related to the performance of their tasks.

Chapter 2, Section 3.1 described that certain properties of an information space such as size, fragmentation, visual homogeneity and complexity of topology can increase the likelihood of users becoming lost and disorientated during exploration activities. A realistic system of source code is generally quite large, highly fragmented and exhibits a dense arbitrary arrangement of semantic relationships and cross-references with no inherent or uniform spatial structure to guide the user. Source code is also visually homogenous.

These properties, individually, and particularly when combined in a single information space, represent a significant potential for disorientation. In fact one could justifiably argue that source code manifests a worst case scenario in terms of scope for disorientation in an information space. A number of properties of source code are particularly inducive to disorientation. These properties, developed from existing research literature, are:

- A high level of fragmentation
- A dense and irregular topology of references and semantic relationships
- Visual homogeneity

3.2.1 Fragmentation

Fragmentation refers to the degree to which a coherent or conceptually related body of information is broken up and stored at non adjacent points in an information space. A high level of fragmentation results in a larger and more topologically dense information network meaning that users need to navigate greater distances and synthesize greater amounts of information during specific exploration tasks. Moreover, the greater separation and displacement of related information makes it more difficult for users to construct an accurate conceptual model and comprehend the information content (Thüring *et al.* 1995). In Chapter 2, Section 2 the discussion of disorientation in hypertext systems highlighted that fragmentation is a significant underlying factor in navigation, task and informational disorientation.

3.2.1.1 Source code fragmentation

Fragmentation is an inherent characteristic of source code resulting from the application of decomposition and abstraction principles in software design and implementation. The practice of separating concerns and decomposing complex systems and operations into cohesive units of implementation generally results in greater flexibility and reusability in a software system (Parnas 1972; Kiczales *et al*. 1997). However decomposition also negatively impacts explorability and comprehensibility by increasing the level of fragmentation in the associated source code (Chu-Carrol *et al.* 2003).

In modern software systems it is common for the source code representing a conceptually coherent portion of system implementation to be broken into numerous discrete program elements which are then spread over a series of source code documents. As a result, exploration and comprehension activities often require a significant amount of navigation between the relevant source code elements and assimilation of a conceptual model from non-adjacent locations in the code space. An activity which is prone to disorientation (Thüring *et al.* 1995). See Chapter 2, Section 3.1 for a discussion of fragmentation and its role in disorientation.

3.2.1.2 Control flow scatter

Chu-Carrol *et al.* 2003 describe fragmentation of source code in terms of 'control flow scatter' because a prominent manifestation of the phenomenon is that the control flow related to coherent portions of system behaviour is heavily in-directed over a series of source code documents and therefore difficult to visualize, examine and reason about as a coherent whole.

Control flow scatter is a pervasive phenomenon which may be considered alongside the evolution of program language design and corresponding decomposition paradigms. As early monolithic systems became difficult to manage due to increasing size and complexity, the concept of procedural decomposition was developed to enable programmers to break up complex program operations into manageable units. Procedural decomposition resulted in significant gains in terms of expressiveness, reusability and manageability in software systems. However, because control flow could be in-directed

arbitrarily in and out of procedure definitions located across various program modules, the process of exploring code became more effortful and cognitively demanding.

The advent of the object oriented paradigm introduced further scope for scatter. In addition to the arbitrary indirection of control flow between method definitions, control flow could be indirected into complex type hierarchies involving inheritance and polymorphism. For instance, in an object oriented environment an abstract method declaration may be 'implemented' by any number of type related methods. Therefore to understand control flow the programmer may also need to explore and comprehend the type structure of the system.

3.2.1.3 Cross cutting concerns

The existence of cross cutting concerns in a system also contributes to source code fragmentation. A cross cutting concern is an aspect or feature of a software system the implementation of which cannot be sufficiently localized within the code space (Harrison & Ossher 93; Kiczales *et al.* 1997). Programming paradigms in general support modularization along a single dimension, a fundamental limitation of modern language technologies referred to as 'the tyranny of the dominant decomposition' (Tarr *et al.* 1999). In a realistic software system, not all concerns can be cleanly modularized using the dominant decomposition. Instead they end up scattered haphazardly across the code space and tangled into the implementation of existing modularized concerns and in some instances other cross cutting concerns (See Figure *3.1*).

Cross cutting concerns contribute an additional layer of fragmentation to source and significantly impinge on explorability, essentially forcing conceptually related portions of code to reside arbitrarily at non-adjacent locations in a code space while providing no interconnecting network of references to guide the programmer.

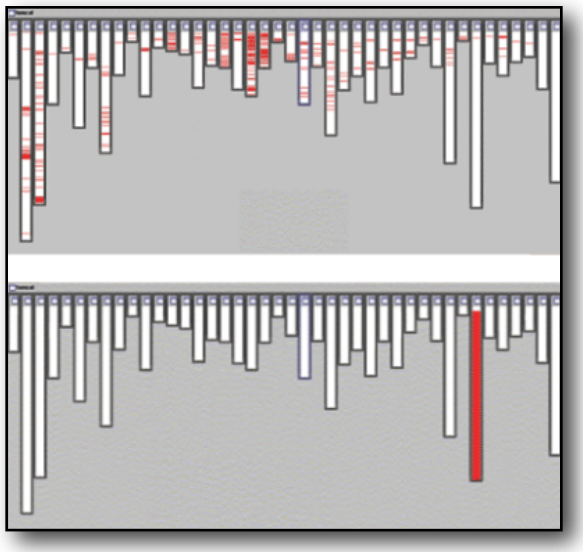


Figure 3.1: A visual representation of a crosscutting concern before modularization (above) and after modularization using AspectJ (below). The code associated with the concern is localized into a single aspect construct which uses point cut definitions to affect base code in a non invasive manner.

3.2.2 Topology

Topology refers to the density and spatial layout of the links connecting units of information to form the structure of an information space. As described in Chapter 2, Section 3.1, a uniform or regular topology helps users to navigate and remain oriented by

reducing route complexity and providing a degree of structural predictability within an information space (Van Dyke Parunak 1989). Furthermore in Chapter 2, Section 2.1 and 2.2 the issue of an regular structure is highlighted as a factor in both navigational and task disorientation.

Certain types of information space support an inherent topological spatialization which can help the user remain oriented while others may be authored to manifest a uniform or predictable structure. A spread sheet, for instance, has an inherent two dimensional structure which may be exploited by user's natural spatial encoding capabilities. For instance if the user knows that a particular collection of data is always located at the top left hand corner of a given sheet they can use this spatial encoding to find said information with minimal effort and thus remain focused on the task at hand. On the other hand the author of a hypertext system (which does not have a natural spatialization) may decide to apply a hierarchal or linear topology to the information network in order to help the user remain oriented during reading and browsing activities. If a hyperspace is too conceptually complex to apply a regular structure, the author may decide to 'cut' or add links in order to simplify the overall topology and prevent the user from becoming lost (Brown 1989).

Source code does not manifest a natural spatialization and programmers rarely have the opportunity to devise or improve topological uniformity within a code space. Due to the sheer size and semantic complexity of software systems, source code generally exhibits an extremely dense and irregular topology of cross-references and semantic

relationships between individual program elements. Figure *3.2* illustrates the potential for topological complexity and density in source code.

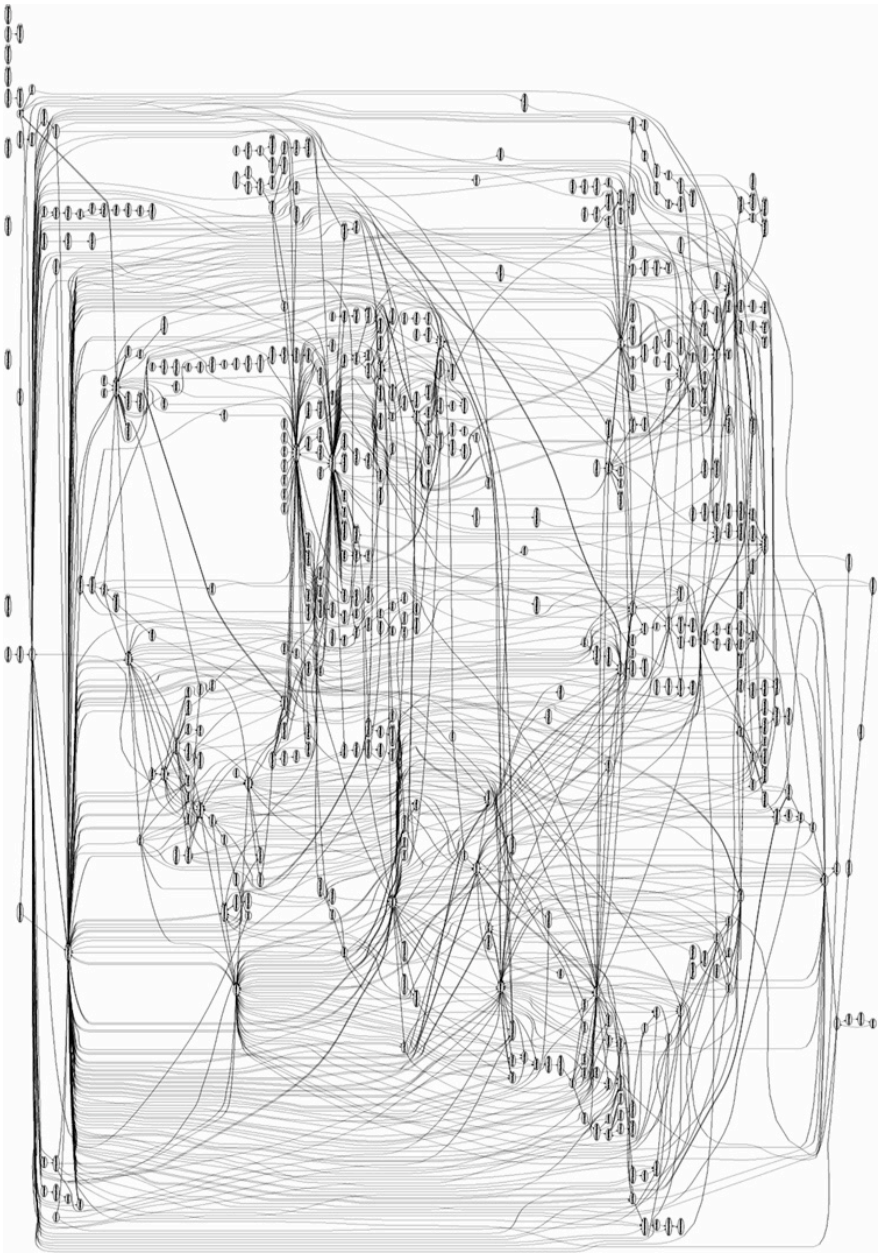


Figure 3.2: A visual representation of fragmentation and topological complexity in source code. This graph (oriented from left to right) depicts the control flow associated with the

generation of a webpage containing a single image in a modern webserver (nodes represent procedure definitions and arcs represent procedure calls).

3.2.3 Visual homogeneity

During exploration activities visual differentiation between information helps users to encode and remember route details and identify familiar information and landmarks within an information space (Kim & Hirtle 1995). Source code is visually homogenous meaning that it can be difficult, at a glance, to distinguish one fragment of source code from another. Furthermore there is little scope for natural landmarks, namely fragments of code which visually stand out and provide orientation cues to the programmer. See Chapter 2, Section 3.2.3 for a description of the concept of the landmark and its importance in helping users remain oriented.

3.2.4 Discussion

The uniquely disorientating structure of source code is largely an unavoidable situation particularly in realistic large scale software systems. The need to express and model highly complex systems and operations in a flat textual space while maintaining engineering metrics such as flexibility, reusability and extensibility invariably leads to a fine grained decomposition and a dense, complex topology of references between program elements. Additionally factors such as sloppy design and implementation, unanticipated evolution and the ‘tyranny of the dominant decomposition’ add further

scope for fragmentation and topological complexity in code space. Source code is also naturally visually homogenous.

In other domains such as hypertext systems, the author of an information space can modify the underlying structure in order to help readers remain oriented. The level of fragmentation can be minimized where possible and a smaller and more uniform topology of links deployed. However, this type of structural engineering is generally not feasible with source code due to its semantic underpinnings. Programmers would need to have the ability to change the structure of the underlying language technology itself which is rarely possible (see Bryant (2002) for related work).

Ultimately the burden of keeping a programmer oriented and focused during software development and exploration activities is the responsibility of the exploration interface in use, which is in most cases an integrated development environment (IDE).

3.3 Exploration in Integrated Development Environments

Source code exploration is generally carried out in the context of an integrated development environment, more commonly referred to as an IDE. An IDE combines a comprehensive set of exploration, development, analysis and debugging tools and facilities into a single integrated and consistent application. This means that programmer's can carry out most aspects of their development and maintenance tasks without the need to repeatedly switch application context.

This section looks at the source code exploration facilities provided by a state-of-the-art modern IDE. The aim is to explore the core interface design and the various mechanisms available for programmers to navigate and explore source code. Using this base of knowledge, the issue of programmer disorientation is discussed, highlighting how certain factors of the interface design contribute to the phenomenon.

The discussion focuses primarily on the Eclipse IDE (Eclipse 2009a). Eclipse was chosen as it represents the state of the art in modern IDE technology. Eclipse has also benefited from a comprehensive effort to address suspected issues related to programmer disorientation (referred to in the IDE community as 'loss of context') (Eclipse 2009b). It was considered acceptable to focus on a single IDE as interface style is generally consistent across modern IDEs. For instance during the research associated with this thesis a variety of modern IDEs were encountered and studied including Eclipse (Eclipse 2009a), Netbeans (Netbeans 2009), IntelliJ IDEA (Intellij 2009) and Microsoft visual studio (Microsoft 2009). All exhibit the same basic structure and provide similar features from a source code exploration point of view. This reflects the relative maturity in IDE interface design and technology.

It should be noted that from this point on when the term 'Eclipse' is used it specifically refers to the Eclipse platform with the Java development toolkit (JDT) (Eclipse 2009c) installed. Eclipse is an extensible IDE platform upon which the JDT provides Java specific tools and features.

3.3.1 Interface basics

The Eclipse interface is based around two high level visual elements, namely editors and views. Editors facilitate the presentation and editing of source code documents (and other text based artifacts such as property and configuration files etc.) while views support exploration, present information and provide an interface construct to IDE tools.

Editors are located in a fixed central editor. Views are 'docked' around the edges of the editor area and may be moved and stacked on top of one another in a tabbed fashion. This allows a variety of views to be visible and accessible to the programmer simultaneously in a single application window (See Figure *3.3*).

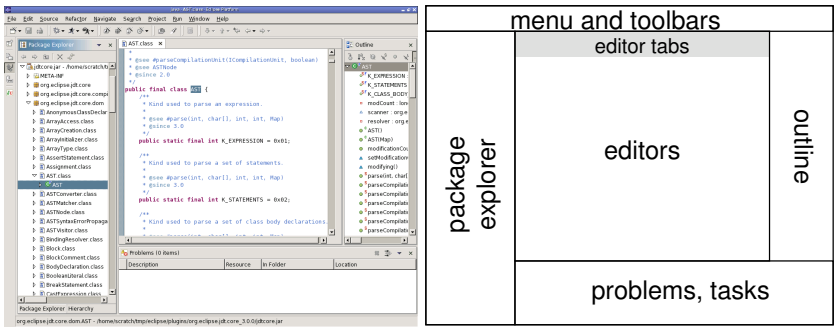


Figure 3.3: The Basic Eclipse IDE interface. The layout is composed of editors stacked in a central editor area surrounded by a collection of stacked views.

Eclipse is 'editor oriented' meaning that each source code document is opened in its own individual editor display which is located in the central editor area (De Alwis & Murphy 2006). The editor area is tabbed so that multiple editor displays may be open at any given

moment. The programmer can switch between the set of open editors using the visible editor tabs running along the top of the editor area.

3.3.2 Source code exploration

Eclipse supports a wide variety of source code navigation and exploration facilities allowing programmers to rapidly traverse and examine the code space during exploration activities. The primary exploration facilities provided by Eclipse are categorized in the following list:

- Hierarchal Project browsing
- Search
- Hypertext style exploration via cross-references
- Index based navigation
- Exploratory views
- Tab based navigation
- Navigation History
- Bookmarks and tasks

3.3.2.1 Hierarchal project browsing

The primary exploratory view in Eclipse is the ‘package explorer’ which is generally located to the left hand side of the editor area. The package explorer presents a

hierarchical overview of the programmers workspace from projects to source folders to packages down to individual source files and program elements. Essentially the package explorer allows the programmer to browse the work space and project structure.

Like all exploratory views in Eclipse the package explorer supports navigation to program elements. If a program element contained in the package hierarchy is double clicked by the programmer, the corresponding program resource or construct is opened and displayed in the central editor area. In the case when an interface, type, method or field is selected, the corresponding source file is opened in an editor, and the fragment of code corresponding to the declaration is scrolled into the visible editor viewport and highlighted to draw the users attention.

3.3.2.2 Search

Eclipse provides a variety of search facilities. The code space may be searched for keywords, or more specifically for particular program elements such as types, methods, fields or interfaces. The programmer can also search for all references to a particular program element or all declarations of a particular element such as an abstract method declaration or an interface type. Search may be initiated from the main Eclipse toolbar, which brings up a dedicated search dialog, or directly on words appearing in the source code editor presentation. This is achieved via context menu options such as ‘find all references in project’ or ‘find all declarations in workspace’.

When a search has been executed, the results appear in the ‘search results’ view, located, by default, below the central editor area. The search results view displays the list

of matches resulting from the most recently executed search. Clicking on an individual match will open the corresponding source code element in an editor display.

3.3.2.3 Hypertext style exploration via program cross-references

The most widely used form of source code navigation in Eclipse is hypertext or 'web' style navigation between related program elements via program cross-references (Murphy *et al*. 2006). As discussed earlier in this chapter, the structure of source code closely resembles that of a complex hyperspace with program declarations serving as information nodes and program references as directed links. Thus hypertext style navigation is a natural application considering the underlying information structure.

Hypertext style navigation is initiated when the programmer holds the Ctrl key and moves the mouse pointer over the visible source code editor display (the use of a modifier key distinguishes between navigation activity and placement of the editor caret). When a reference is encountered under the mouse, such as a method, type or field identifier, the associated word of source code transforms into a hyperlink, adopting the traditional underlined blue font to indicate that navigation is possible. When the hyperlink is activated, the corresponding source code declaration is opened and highlighted in a source code editor display.

3.3.2.4 Index based navigation

Eclipse supports indexing into code space based on type or resource name. From the main application window the programmer can call up the ‘open type’ or ‘open resource’ dialogs using specific key combinations. The name of a type or resource is then entered into the dialog and all matching artifacts in the system are displayed as an alphabetically ordered list. The programmer can click on a particular match and the corresponding type or resource is opened in an editor display.

Index based navigation in Eclipse allows programmers to navigate directly to types and resources in a code space that are known to exist, without exhaustive exploration, searching or browsing via exploratory views. The system is particularly usable due to support for wild cards and incremental evaluation of matches. The programmer can search for a type or resource when only a portion of the name is known and can evaluate numerous permutations of a given wild-card query without leaving or reopening the dialog.

3.3.2.5 Exploratory views

Most other exploratory navigation in Eclipse is facilitated by the various exploratory views which come bundled with the JDT. The type hierarchy view provides a hierarchical overview of the type structure associated with a particular class in the system, this includes super class, implemented interfaces and sub classes. A sub pane in the hierarchy view presents a detailed summary of all fields and methods declared locally and inherited

from the type hierarchy. The call hierarchy view allows the programmer to view all callers and callees of a particular method definition and to navigate control flow in either direction. Finally, the outline view displays a compact outline of the currently visible source code document (the contents of the focused source code editor). The outline view displays imports, fields, methods along with status such as visibility and signature. When the outline view is open it automatically updates itself to the currently focused source code editor instance.

3.3.2.6 Tab navigation

During a particular development or maintenance task the programmer can flip between the set of open source code documents using the editor tabs located across the top of the editor area. Tabs are arranged in first opened order from left to right. When the number of open source code documents exceeds the available tab space, the programmer can use a drop down panel to view and filter all open documents as an alphabetically ordered list.

3.3.2.7 Navigation history

To facilitate back tracking and revisiting of previous locations, Eclipse maintains a record of the programmer's navigation history. Navigation history is stored as a list of source code locations ordered chronologically with duplicated entries removed. The programmer can move back and forward through navigation history using buttons located in the main Eclipse toolbar. History can also be viewed and indexed as a drop down menu containing

a list of filenames. The maximum number of entries in the navigation history is 50. If this window is exceeded, the oldest entries in the list are discarded. Because navigation history is one dimensional, forward history is cleared when a location is visited which deviates from existing forward history. For instance, if the programmer backtracks from a given location and then pursues an alternative route the backtracked route is discarded in favour of the newly pursued route. Eclipse does not provide support for multiple routes or digressions in its navigation history.

3.3.2.8 Bookmarks and tasks

Eclipse supports the notion of bookmarks. A programmer can select a particular source code location or fragment of code and 'bookmark' it using a context menu action. The bookmarked location is then recorded, and appears in the bookmarks view for perusal at a later date. The bookmarks view supports navigation to bookmarked locations allowing the programmer to navigate to pertinent locations in the code space without the need to browse or search.

Eclipse also supports the concept of tasks which are very similar in nature to bookmarks but have additional task related state such as priority and a completion flag. Tasks may also be automatically recorded via specific comment patterns such as 'TODO' and 'FIXME'. The programmer can simply create a comment containing a predefined (or custom defined) keyword pattern and Eclipse will automatically enter a corresponding task in the tasks view.

3.3.3 Programmer disorientation in the IDE

The study of programmer disorientation during source code exploration activities in the IDE setting is still an emerging area of research, and literature pertaining to the topic is rather space.

Janzen & De Volder (2003) describe that programmers often become disoriented when switching between exploratory IDE views (for instance the package explorer, hierarchy view, search results view etc.) due to fragmentation of exploration history across a set of views, which eventually leads to a loss of context and consequently disorientation. Kersten & Murphy (2005) explain that programmers suffer from information overload when switching task focus due to a lack of explicit task state in the IDE. Elves (2005) describes that programmers often become 'altogether lost' when navigating the complex web of references and relationships between source code documents.

The most significant recent work on disorientation in the IDE is provided by De Alwis & Murphy (2006). Based on informal reports of programmers becoming 'lost' and disoriented during software development and exploration activities, the authors carried out an exploratory field study in an attempt to gain a greater understanding of the phenomenon. A group of professional software developers were observed, on site, for a number of hours as they carried out their daily development tasks using the Eclipse IDE. The effort resulted in a characterization of programmer disorientation and more significantly, identified a number of factors (many of which are concerned with interface

design) which contribute to disorientation during source code exploration and development activities in the IDE.

De Alwis & Murphy (2006) characterize programmer disorientation as a combination of navigational disorientation and task disorientation, both of which are deeply inter related.

Navigation concerns the decisions and actions which facilitate a programmers coherent movement and transfer of focus through a system of source code. Navigational disorientation occurs when a programmer:

- has lost their sense of location and direction in the code space (What am I looking at?)
- is unable to locate information of interest (Why cant I find?)
- has lost their recent history, is unable to remember how they came to their current or past locations (What was I doing?)

Task disorientation is associated with the programmers goals and intent as they explore a system. Task disorientation occurs when a programmer:

- cannot remember their intent having arrived at a location (What was i going to do?).
- has pursued, or was distracted by, an alternate problem and has failed to return to the original task

The characterization of programmer disorientation proposed by De Alwis & Murphy (2006) is reminiscent to that of disorientation in the hypertext domain (see chapter 2). Programmers suffer from the same fundamental spatial and navigational disorientation in code space as well as issues managing multiple threads of concentration (embedded digressions) and maintaining ongoing intent. Interestingly, informational disorientation is explicitly excluded from the characterization. The authors highlight a distinction between conceptual disorientation and spatial disorientation within an information space (Mayes *et al.* 1990), concluding that a degree of conceptual disorientation is to be expected during intense knowledge discovery and refinement activities such as source code exploration. Therefore, while conceptual disorientation is acknowledged it was not considered to be a significant part of the characterization.

De Alwis & Murphy (2006) also identify a set factors which they believe contribute to the incidence of programmer disorientation. These factors are based on an analysis of the various incidents of disorientation observed during their study.

Factors inducive to programmer disorientation (De Alwis & Murphy 2006) include:

- A lack of navigation history
- Thrashing to obtain context
- A lack of support for the pursuit of digressions
- Problems associated with a lack of code familiarity

3.3.3.1 A lack of navigation history

As previously discussed in this Chapter (3), Section 3, modern IDEs support a wide variety of exploratory navigation facilities enabling programmers to rapidly navigate between source code documents and elements located throughout a system of source code. A programmer can browse the project structure (See Chapter 3, Section 3.2.1), navigate via program cross references (See Chapter 3, Section 3.2.3), search for particular keywords of program elements (See Chapter 3, Section 3.2.2 and 3.2.4), traverse the history stack (See Chapter 3, Section 3.2.7), leverage exploratory views (See Chapter 3, Section 3.2.5) and make use of bookmarks and task definitions (See Chapter 3, Section 3.2.8). The ease of motility facilitated in modern IDEs is essential due to the inherent fragmentation and topological complexity of code space (See Chapter 3 Section 2).

However Eclipse, like all other IDEs that were studied during this research, effectively restricts the programmer to a single editor display at any moment during exploration activities. This is primarily due to screen space limitations. The situation epitomizes the ‘keyhole property’ discussed in Chapter 2, Section 1, that being a spatially sizeable information space (the set of source code documents/elements making up the system) is examined via a limited aperture (the visible viewport of the focal editor display).

As a programmer navigates from location to location within a system of source code, the visible editor display is continuously replaced to reflect the current or focal source code location. There is no visible indication of how the programmer arrived at the current location or how it relates to previously visited elements and the structure of the

surrounding system. Essentially, source code is explored as a series of isolated source code editor displays, each replacing its predecessor and the programmer internalizes the burden of maintaining exploration context and orienting to each new display as it appears on the screen. This situation is indicative of a significant lack of visual momentum in the exploration interface. See Chapter 2, Section 3.2 for a discussion of visual momentum.

During their study, De Alwis & Murphy (2006) observed that programmers had difficulties remembering the path leading them to their current source code location and consequently the reason as to why they had come to arrive there. To regain context programmers were observed to backtrack to previously visited locations, recovering the relationships that brought them to the current location in an attempt to rebuild context and refresh their memory of the intent being pursued. If this strategy failed programmers were observed to close all open editor displays and restart their task from a familiar location in the code space.

3.3.3.2 Thrashing to obtain context

Thrashing (Henderson & Card 1986) is a behaviour associated with disorientation which is commonly associated with keyhole displays (Watts-Perotti & Woods 1999) such as that presented by modern editor oriented IDEs. When a user needs to correlate, compare or contrast information located at disjoint points in an information space, which cannot be brought together in a single display due to interface constraints, they tend to repeatedly navigate or flip between displays containing relevant information in order to gain the necessary overview to accomplish their goal.

During their study De Alwis & Murphy (2006) observed programmers repeatedly scrolling and jumping within and between source code editor displays. Thrashing activity requires the programmer to store additional information in working memory (the context being shared across the different views) and focus attention on interface manipulation activities as opposed to their underlying task. This kind of extraneous activity may be distracting enough for the programmer to lose their focus and become disoriented.

3.3.3.3 A lack of support for pursuit of digressions and insufficient task context

A digression occurs when a programmer suspends their current task or intent to pursue or is distracted by another, perhaps unrelated task or intent. A digression may in turn spawn further digressions and eventually the original task or intent may be forgotten. For instance a programmer might receive an email from a colleague while investigating a bug they have noticed in the system. The disruption results in the programmer focusing on replying to the email for a given period of time. Once finished the programmer may no longer remember their original bug related task and neglect to resume their work on it. Alternatively the programmer may remember the original task but may have forgotten most of the associated context which then needs to be rebuilt, requiring additional time and effort.

Because there is no explicit manifestation of task context in the IDE the burden rests on the programmer to manage suspended tasks in memory and recreate the context of suspended tasks once resumed. During the study, programmers were observed having

problems with digressions, often pursuing digressions without recording the original task and thus failing to resume the suspended task at a later time.

Digressions may also occur at a more fine grained level during source code exploration activities (see Embedded digressions (Foss 1989a) in Chapter 2, Section 2). The act of investigating source code results in a continuous stream of small scale digressions, which could also be described as threads of thought (De Alwis & Murphy 2006). For instance when investigating a piece of code, the programmer may be obliged to investigate an associated piece of code in another area of the system and so on. Eventually, due to a lack of explicit navigation context, the programmer may eventually forget their original intent which may cause important information to be overlooked.

3.3.3.4 Code familiarity

During the study participants reported that the exploration of unfamiliar source code was a significant cause of disorientation. Participants described that developing a mental model of unfamiliar code was difficult, and that they would often loose track of their location and become lost in unfamiliar areas of the system.

3.4 Mitigating programmer disorientation

Programmer disorientation is a significant concern in modern IDE design. Recovering from a state of disorientation takes both time and effort which has a negative impact on overall programmer productivity and satisfaction.

Modern IDEs provide various features designed to help the programmer remain oriented during exploration activities, and to reduce the effort associated with the exploration of fragmented source code. Moreover a number of research tools have also been proposed and developed to address particular aspects of the phenomenon. The distinction between alleviating disorientation and remaining oriented isn't hugely important and is mostly a question of terminology. Tools designed to alleviate disorientation essentially help the programmer to remain oriented. And recovering from disorientation is a secondary point when a system is designed to help the programmer remain oriented.

3.4.1 Core IDE technologies

The Eclipse IDE supports the concept of pop-ups which are essentially a form of display overlap (an approach to increase visual momentum (Watts-Perotti & Woods 1999) see Chapter 2, Section 3.2). From a given source code editor, the programmer can position the mouse cursor over a source code reference for a short period of time to invoke an overlay display containing a read only copy of the corresponding source code declaration. This pop-up technique allows the programmer to view remote source code declarations in the context of a source reference and thus avoid navigating away from the current location and suffering from the associated loss of context. However the approach is limited as pop-ups tend to occlude a considerable portion of the source context and generally cannot be moved by the programmer to prevent such a scenario. More importantly, the pop-up display does not support any further exploration and thus the

programmer is limited to a single level of surrounding context, if the programmer wants to consider a further level of context they have to leave the current source code location and deal with the resulting loss of navigational context.

A second technique used in Eclipse is the declaration view which is similar to a source code pop-up but is located in a fixed view outside of the editor display, thus avoiding the occlusion problem. The declaration view, typically docked below the central editor area, allows the programmer to examine the source code declaration associated with a selected program reference from the editor display. Again the programmer can view surrounding context without navigating away from the current source code location. However, the declaration view requires large visual saccades away from the source context and is also, like the pop-up, restricted to a single level of surrounding context. It is also the case that programmers tend to avoid additional views in the IDE (De Alwis & Murphy 2006). Additional views take up precious screen real estate in an already cramped visual space.

Eclipse also supports the use of multiple editor displays which may be used to circumvent the need to thrash during exploration activities. Editors may be tiled in the editor area so that a number of source code locations can be be made visible in a simultaneous fashion. To use this feature the programmer drags editor tabs into the editor area in such a way that they begin to tile in the desired manner. While this mechanism may allow the programmer to view multiple source code displays, it also reduces the space available for each individual display. The programmer may need to carry out a considerable amount of interface adjustment (scrolling and resizing) to provide enough

visible context within an individual display to make the code readable and the approach worthwhile.

The navigation history stack See (Chapter 3, Section 3.2.7) may be considered as a tool which allows a programmer to recover from a state of disorientation. When a programmer gets lost in the code, they may use the history stack to back track to previous locations in order to recover their context. However the history stack is still operating over the keyhole display and as such often facilities disorientation on its own behalf.

Finally the index based navigation facilities in modern IDEs (Chapter 3, Section 3.2.4) allows programmers to navigate directly to types and resources in a code space that are known to exist without exhaustive exploration, searching or browsing via exploratory views. This mechanism is valuable when the programmer knows the various types in the system but is less useful during open exploration activities.

3.4.2 JQuery

Janzen & De Volder (2003) maintain that when exploring source code in the IDE programmers suffer from disorientation when switching between exploratory IDE views. As discussed in the previous section, IDEs support a variety of exploratory views. However, exploratory views are typically tailored specifically to support exploration of a particular type of program relationship. For example a type hierarchy view will allow the programmer to explore along inheritance relationships while a package explorer view will support the exploration of project structure. These views usually provide a tree like interface where nodes represent program elements and sub nodes are related to parents by

a specific relationship type. The advantage of the hierarchical structure is that a visible representation of structure and exploration context is available which helps the programmer to remain oriented and focused on their task.

However, because each view is limited to a particular type of program relationship, when a programmer wants to explore using an unsupported relationship type they need to switch to a different view. Janzen & De Volder (2003) claim that the process of switching between exploratory views is disorienting in itself, but moreover the programmer's exploration path becomes fragmented across multiple separate views. As a result the programmer can lose track of their exploration context and become disoriented.

In response to this problem, the JQuery tool (Janzen & De Volder 2003) was developed (See Figure *3.4*). JQuery is a source code exploration tool that combines aspects of a hierarchical browsing tool and a software query tool. The aim of JQuery is to reduce the mental burden associated with source code exploration by:

- Providing a coherent and unfragmented representation of the programmers exploration path which helps the programmer to remain oriented in the exploration task

- Allowing the programmer to explore a broad range of program relationships from a single IDE view thus reducing the disorientation associated with view switching

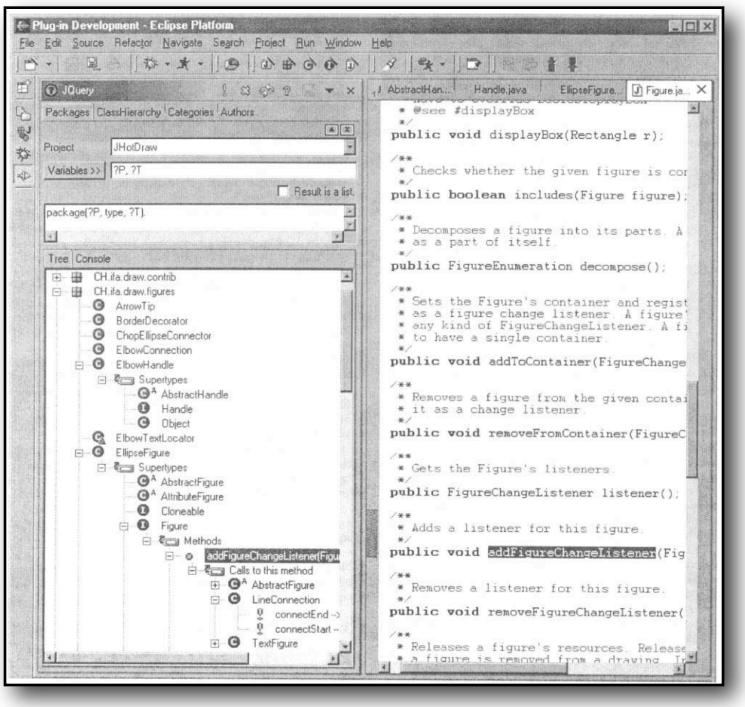


Figure 3.4: The JQuery view (left).

The programmer uses the JQuery tool by entering a query manually via a query language or, more commonly, selecting from a set of predefined queries built into the tool. The results of the query are then displayed in the JQuery tree view. Associated with each node in the view is a specific context menu which lists the ways in which the tree can be expanded at that node. The tool provides a comprehensive suite of default program relationships and custom relationship types may be added to the menu structure by the user via a configuration file.

JQuery allows a programmer to explore a software system via a comprehensive set of program relationships from within a single hierarchical view. The programmer avoids the requirement to switch between a variety of separate views and the explicit unbroken representation of the exploration path allows the programmer to see how a

particular source code location or element fits into the overall exploration context and supports backtracking and the perusal of multiple exploration paths in parallel. The JQuery too was not evaluated via a user experiment to determine its effectiveness at alleviating programmer disorientation.

3.4.3 NavTracks

Elves (2005) explains that programmers are prone to getting lost and suffering from disorientation when navigating large complex software systems. When working on a given development or maintenance task, a programmer will often need to repeatedly examine and/or modify source code contained in a number of task relevant files spread across the code base. Over the course of the task, the programmer will need to remember the set of relevant files and how to navigate through the information space to locate them when necessary, but source code is often structurally complex and the constant navigation between files is a cognitively draining process. The programmer might occasionally get lost and lose track of their intent when navigating through the software space.

NavTracks (Singer *et al.* 2005) is a source code navigation tool designed to alleviate this problem by recommending related files to the user (See Figure *3.5*). NavTracks interactively analyses a programmer's navigation history and records associations between visited files. These associations are then used as the basis for recommending potentially related files as a developer navigates the system. The goal of NavTracks is to present a user with an accurate list of related files when working on a

particular file. Instead of having to recall the next file and then navigating to it, the programmer simply recognizes (a cognitively cheap process) and selects the related file in the NavTracks view and continues undisturbed with their task.

The main principle behind NavTracks is that exploration paths through an information space can reveal the user's model of how information should be connected thus reflecting the user's mental model of the system.

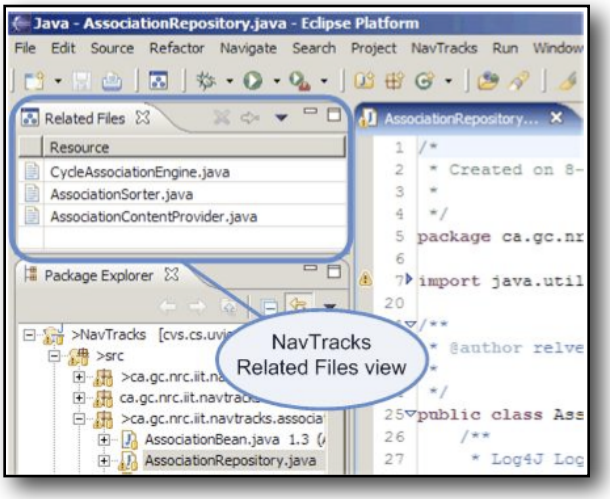


Figure 3.5*: The Navtracks view displaying related source files associated with the current source file. Related files are inferred based on the programmers navigation history.*

3.4.4 Mylar/Mylyn

Mylar (Kersten & Murphy 2005) is similar in concept to NavTracks but operates by filtering IDE view as opposed to recommending related files. Again if one considers a

programmer working on a large complex software system. Over a period of time the developer will work on a variety of tasks and during each task will visit and revisit multiple task relevant source code locations. IDE views are intended to help the programmer to locate, examine and navigate between these locations. However in large software systems, the utility of IDE views degrade due to information overload. The proportion of task relevant information in a view decreases and the programmer spends more time searching IDE views and navigating for task relevant code. The Mylar tool (See Figure *3.6*) is designed to mitigate this problem by filtering views for task relevant information.

Mylar monitors a programmer's interaction patterns within the IDE and builds a degree of interest (DOI) model (Card & Nation 2002) for program artefacts. When a programmer visits or edits a program artefact its DOI value is increased. The DOI will then degrade over time if the artefact is not revisited by the programmer. The DOI values are applied to a filtering function which operates over the IDE views so that only task relevant artefacts are displayed to the programmer. More interesting or relevant artefacts (those with higher DOI values) are also highlighted while less relevant artefacts are displayed in a less prominent manner or filtered out altogether.

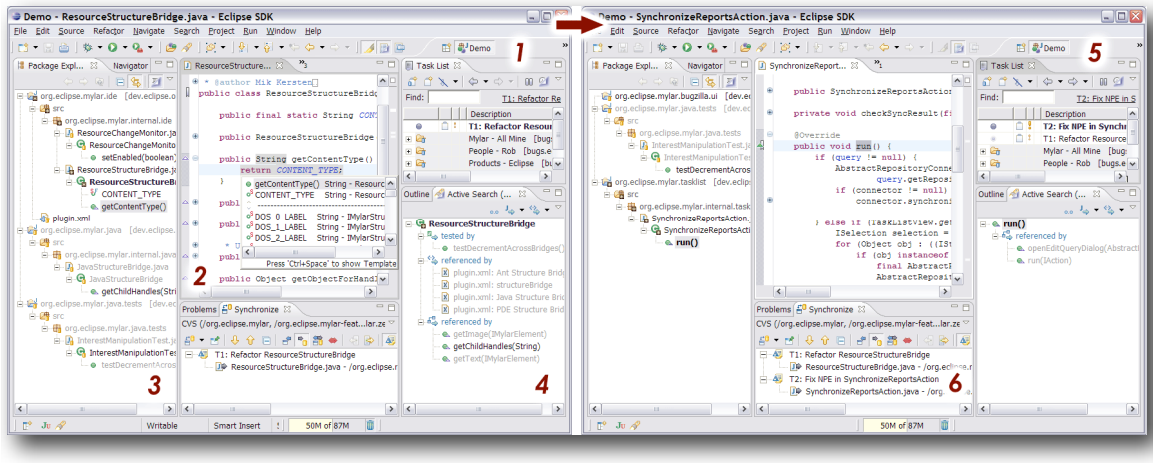


Figure 3.6*: Mylar: before and after, view filtering based on DOI.*

A more mature version of Mylar entitled Mylyn (Eclipse 2009e), which is available as an extension of the Eclipse IDE, includes task context as a first class entity. The programmer can create and switch between tasks each with their own particular DOI which is applied to the set of IDE views.

3.5 Summary

Source code manifests a particularly disorientating information space due to a variety of inherent structural and organizational properties such a high level of fragmentation, a dense and irregular topology of references and semantic relationships and a homogenous appearance. Based on current language technology, it is generally unfeasible to modify the structure and layout of source code in order to bring about a more regular and less disorientating information space. This is a common approach to reducing disorientation in the hypertext domain (Brown 1989; Van Dyke Parunak 1989). Instead it is the role of

the IDE, the medium in which source code exploration is generally carried out, to help programmers deal with the complex and fragmented structure of code while remaining focused and oriented during their exploration and development tasks.

Modern IDEs, while providing a wide gamut of navigation and exploration facilities to support the exploration of source code, exhibit certain interface design characteristics which, combined with the disorientating nature of the underlying information space, significantly contribute to the incidence of programmer disorientation:

- Modern IDEs generally exhibit a keyhole style interface which effectively restricts the programmer to viewing a single fragment of source code at any particular moment during exploration activities.
- Moreover there is little to no visual momentum between successive source code displays, no visible navigation or task context and no explicit representation of the programmers evolving conceptual model.
- In addition, the support for examining multiple fragments of code in a simultaneous fashion tends to rudimentary, limited and generally problematic leading to thrashing problems and difficulties comprehending code.

The result of these interface design issues is a considerable lack of cognitive support for the programmer as they carry out source code exploration activities. Essentially, the programmer is forced to maintain a large amount of context in their working and short term memory which is prone to distraction and overloading, particularly when faced with

interface problems such as getting lost in the code space, thrashing between displays and engaging in interface adjustment activities.

A number of approaches have been designed to tackle programmer disorientation in the IDE. Standard IDE technologies such as pop-up displays and the Eclipse 'declaration view' allow programmers to explore secondary relationships in source code without context loss associated with explicit navigation. However this support is very limited and as such often goes unused (De Alwis & Murphy 2006). Researchers have also developed a number of approaches. Janzen and De Volder (2003) developed the JQuery tool which allows programmers explore source along a variety of relationship types in a single exploratory display. This means that the programmer avoids switching between exploratory views and has a coherent overview of their exploration task in the single JQuery view. Elves (2005) developed Navtracks a tool which monitors the programmers navigation patterns and recommends related files, however the programmer is . Finally Kersten & Murphy (2005) developed the Mylar approach which introduces task based filtering into IDE views and promotes the task as a first class IDE entity.

Chapter 4

Inline Source Code Exploration

A potentially significant approach to alleviating programmer disorientation during source code exploration activities in the IDE is the use of inline exploration. Inline exploration is a mechanism for exploring an inter-linked non linear information space, such as that presented by source code, in a cognitively supportive and contextual manner. The primary tenet of the approach is that instead of explicitly navigating from one isolated display of information to the next, which results in a continual loss of context over a series of navigational transitions, the user progressively introduces related fragments of information into the context of a focal, or primary, information display.

There are a number of attractive properties of inline exploration which are relevant to the various interface problems underpinning programmer disorientation in modern IDEs. The process of inline exploration results in a visible and manipulatable representation of navigation history and context as the user progressively ‘expands’ into the information space related to a particular information display. Context loss is also

avoided and with it the need to reorient to each display as it appears on the screen. Inline exploration also supports the simultaneous examination of multiple fragments of related information which may supersede the need to thrash between individual displays, and supports the comprehension of highly fragmented information in a coherent and unified manner. The approach also supports the pursuit of multiple exploratory digressions while maintaining the originating context which can potentially ease problems associated with embedded digressions.

The primary thrust of this research is to evaluate the effectiveness of inline exploration as a means of exploring source code in a more cognitively supportive and less disorientating manner. This work has the potential to significantly improve programmer productivity and satisfaction and also inform development of IDE tools and technologies aimed at alleviating programmer disorientation.

4.1 Origins and related work

Inline exploration has yet to be coherently identified and presented as a mechanism for exploring source code in context with the broader aim of alleviating disorientation problems. Instead the concept has been synthesized from a number of projects and research efforts which exhibit various aspects of inline and contextual exploration. This section describes the relevant background work associated with the concept of inline exploration.

4.1.1 The guide hypertext system

Guide (Brown 1989) was an early hypertext system pioneered by Peter Brown at the University of Kent at Canterbury in 1982. The system was actively developed in various incarnations throughout the remainder of the decade, both as a platform for ongoing hypertext research, and a commercially available product.

4.1.1.1 Motivating factors

At the time when Guide was under development hypertext systems generally supported the traditional ‘out of line’ exploration mechanism still predominant in todays world wide web - and also the basis for source code exploration in modern IDEs. The user would explicitly navigate from one frame or page of information to the next via links embedded in the information display, with each new display or frame of information replacing its predecessor on the screen or appearing in a newly spawned window. Simultaneously, the concept of becoming ‘lost in hyperspace’ was also emerging as a significant concern during this period of hypertext research (Brown 1989; Conklin 1987). Hyperspaces were increasing in both size and topological complexity and consequently pushing the usability limits of the rudimentary exploration mechanisms in use.

The prevailing wisdom was that a visual overview map of the hyperspace structure was necessary to prevent users becoming lost during reading and browsing activities, some authors even describing it as a pre-requirement to hypertext browsing tools (Halasz 1988). However the Guide system pursued a very different approach,

tackling the issue in terms of the core exploration mechanism as opposed to providing an extraneous overview map. The Guide interface was designed to ‘distance’ or abstract the user from the underlying directed graph structure representing the hyperspace. This particular aim was achieved via an ‘in-situ’ exploration technique.

4.1.1.2 In-situ exploration in Guide

The Guide interface presents the user with a single ‘scroll’ of information embedded within which are interactive hyperlink-like elements referred to as ‘replace buttons’ (See Figure *4.1*). A replace button appears as a standard fragment of text but is distinguished from surrounding information via a bold font. When the user selects a particular replace button it is replaced, in situ, with a portion of information associated with the button (See Figure *4.2*). This new information may contain nested replace buttons thus facilitating nested replacement, and essentially, in-situ exploration.

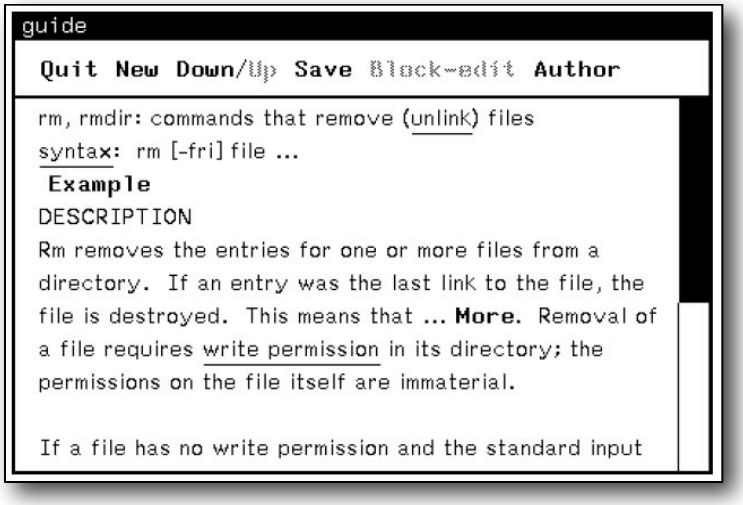


*Figure 4.1: The Guide hypertext system, replace buttons '**Example**' and '**More**' are distinguished with a bold font.*

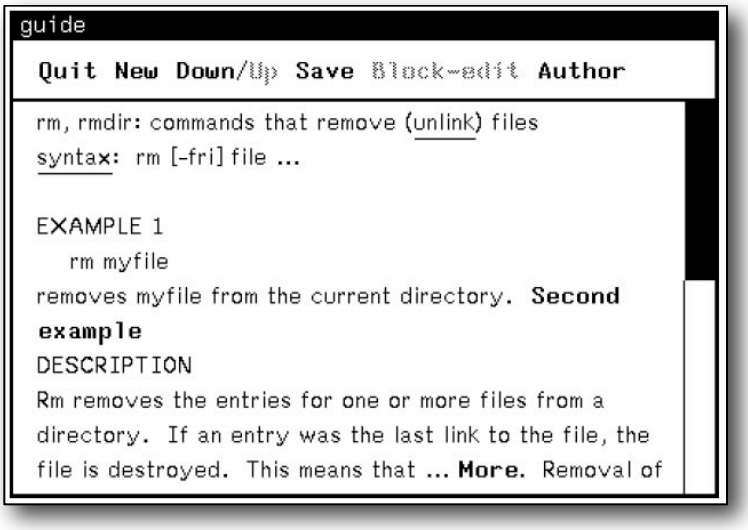


*Figure 4.2: The result of selecting the '**Example**' replace button, the button has been replaced with additional information and a further nested replace button entitled '**Second example**'.*

Replace buttons are local constructs contained within a discrete hyperdocument which support introduction of locally defined information. However the ‘usage button’ extends the basic concept of the replace button to allow for the introduction of cross-referenced information, i.e. information located within a remote hyper document.

The ‘usage button’, which is indistinguishable from a replace buttons at the user level, acquires its information by following a cross-reference link to a remote hyper document. The hyperspace author defines a particular range within a hyper document and the usage button uses this definition in order to extract the necessary chunk of information. When the user selects a usage button, the remote definition is copied and used to replaced the source usage button.

4.1.1.3 Multiple copies

The existence of information from remote hyperdocuments out of their native context necessitates the management of multiple copies. The Guide system treats multiple copies of a particular definition as independent of its native counterpart. If the original definition is edited (Guide supports both the editing and exploration of hyper documents), the changes are not propagated to any copies, they essentially become stale. Furthermore, edits to in-situ copies are transient and lost during structural saves of the overall hyperspace.

4.1.1.4 Discussion

Guide introduced the basic concept of interactively replacing portions of a digital document (usage and replace buttons) with information copied from a remote document - as a viable alternative to explicit navigation between discrete displays. The idea was that the user would be abstracted from the fragmented and topologically complex nature of the information space and thus would avoid getting lost.

The in-situ exploration mechanism pioneered by Guide was not evaluated via a user experiment or study, thus the merit of the approach has never been fully verified beyond its natural resonance.

4.1.2 The Fluid Document Project

Another significant piece of work related to inline exploration is the fluid document project. The fluid document project (Zellweger *et al.* 1998) was carried at Xerox PARC over a number of years and explored user interface techniques for dynamically incorporating related information into the context of digital documents. The broad aim of the research was to minimize the distraction and context loss suffered during a readers shift in focus from a primary information context to a secondary or related information context - a common theme during digital reading activities.

The term ‘fluid‘ was applied to highlight a lightweight, contextual, and animated approach to information access, allowing the reader to fluidly shift attention from

primary to related material and back again with minimal disruption and cognitive overhead.

4.1.2.1 The Fluid UI

The Fluid UI (Zellweger *et al.* 1998), which constitutes the core tenets of the fluid document approach, consists of three basic interface design principles for secondary information access - visual cues, animated transition and accommodation.

The Fluid document interface design principles (Fluid UI):

- Visual cues
- Animated transition
- Accommodation

A visual cue is an annotation embedded within a primary information representation - for instance a digital document - which indicates the existence of linked secondary or related information. A cue can be textual or graphic depending on application and user experience preferences. Examples include hyperlink style underlines or small interactive shapes and images embedded directly in the body of a digital document.

The reader interacts with the visual cue in order to trigger the exposure or introduction of secondary information, referred to as ‘glosses’. Interaction style can be light weight such as simply hovering over a cue with the mouse or a more conventional such as clicking. The transition from a cue to the exposure of secondary information is

generally animated in order to help the reader track changes and minimize disruption during reading activities (Zellweger *et al.* 2000).

The topology/layout of the primary information space or document is dynamically altered in order to 'accommodate' the introduction of secondary material. This allows secondary material to be presented in a readable format while still retaining the context of the primary material.

4.1.2.2 Introduction techniques

Subject to the fluid UI principles a number of techniques were designed to facilitate the introduction of glosses (units of related information) into the context of digital documents.

The most basic technique for introduction is 'interline expansion'. The interline expansion technique causes the primary material to 'split' horizontally at the annotated line and the gloss is then introduced into the newly acquired space. In the example depicted in Figure *4.3*, visual cues are represented as underlines, expanded glosses are presented in red. The interline expansion technique results in related information remaining close to the surrounding primary context. In scrolling displays, the interline expansion technique allocates the required additional space at the expense of pushing the necessary amount of primary material out of the visible viewport and thus increasing the overall scroll distance. For fixed size and page based displays, extra space is acquired by either reducing the interline space between all lines on the page or "squashing" the text above and below the gloss by reducing its font size.

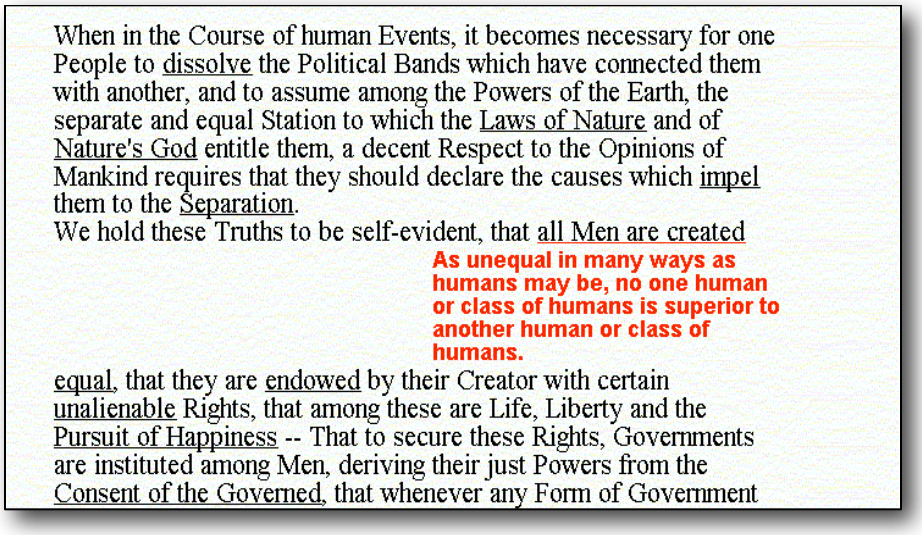


Figure 4.3: The interline expansion technique. The primary document is split horizontally and related information introduced inline into the newly available space.

Another technique, entitled margin callout (See Figure *4.4*), avoids topological alteration of the primary material by placing the gloss in available white space on the page margin. When a cue is activated, an animated line extends in real time from the cue out to the nearby margin where the gloss is expanded. An attraction of the margin callout technique is that the original document remains unaltered in the presence of glosses. Although margin callout sacrifices close proximity between the cue and the expanded gloss, animation is applied to gradually draw the reader's eye to the gloss and back to the source context when the gloss is dismissed.

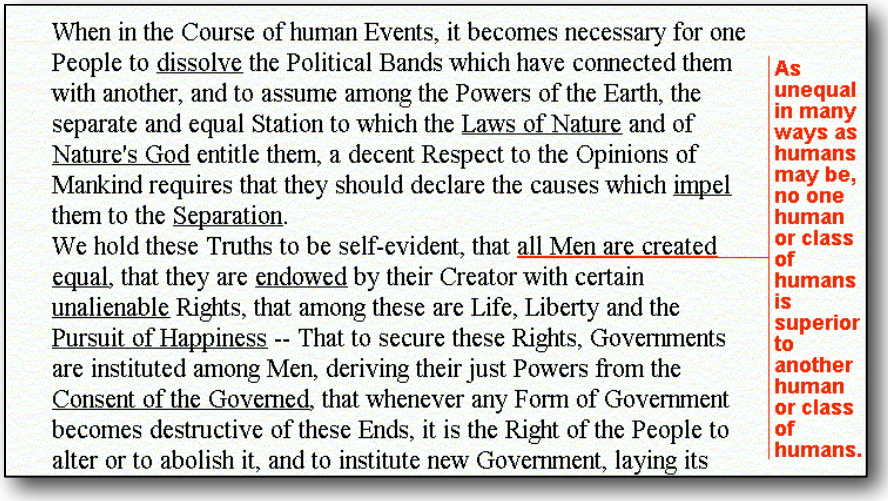


Figure 4.4: The margin call-out technique, information is introduced into the margin associated with the primary document. Introduction is animated to draw and retract the users attention.

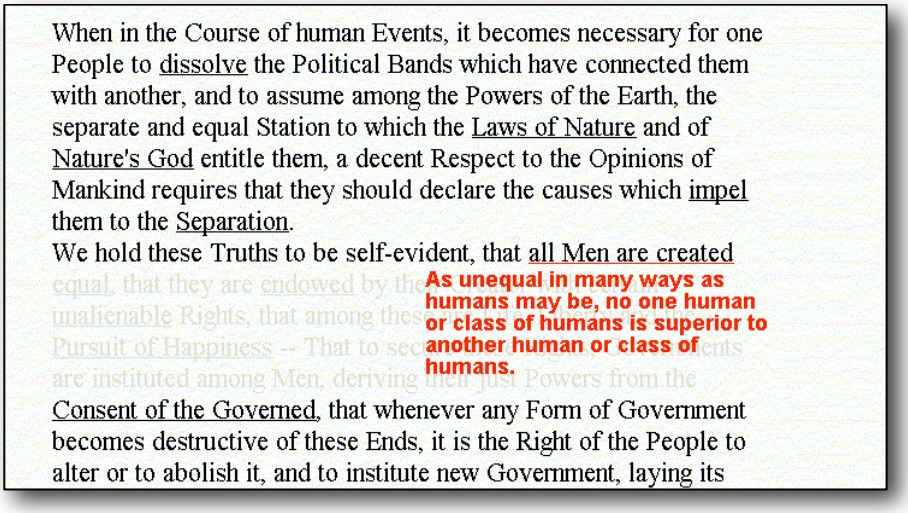


Figure 4.5: The fluid overlay technique. A sufficient amount of primary material is faded to allow secondary material to be overlaid in context.

The final introduction technique is entitled fluid overlay (See Figure *4.5*). The fluid overlay technique is similar to the interline approach, however the gloss is not introduced

into space made available by altering the topology of the primary document, instead the portion of the primary document under the cue is faded and the gloss is placed directly over it. The occluded portion of the primary document remains visible enough to allow a certain degree of readability simultaneously with the introduced information.

4.1.2.3 Nested introduction

Fluid documents support the concept of nested visual cues and thus nesting of glosses. Nesting facilitates the expansion of glosses within the context of already expanded glosses allowing a form of inline or fluid exploration. In the example illustrated in Figure *4.6*, a number of glosses are nested in a single display. To achieve this degree of exploration using a standard display the user would need to perform multiple display switches.

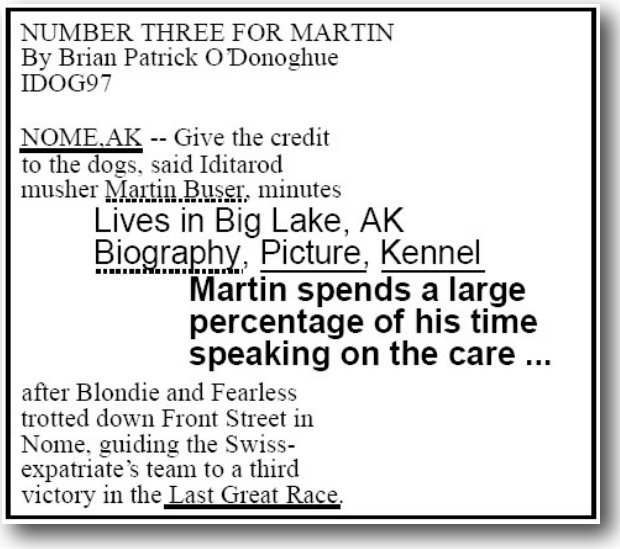


Figure 4.6: Nested visual cues and glosses. This figure depicts the introduction of glosses into existing glosses.

4.1.2.4 Applications of fluid document technology

The initial implementations of the fluid document techniques were based on simple text documents and hypertext. The fluid links hypertext browser (Zellweger *et al.* 1998) used fluid document techniques in order to fluidly reveal a summary of the destination of hypertext links in order to help the reader decide whether or not to follow the link without leaving the source context and suffering the corresponding cognitive overhead.

With both the text based implementation and the fluid links browser glosses were expanded in a lightweight fashion by the user moving the mouse over a visual cue. When the mouse left the area occupied by the visual cue the gloss retracted. The concept of "frozen" glosses was also introduced. This feature allowed the user to freeze a number of glosses in an expanded state at any one moment for a given document.

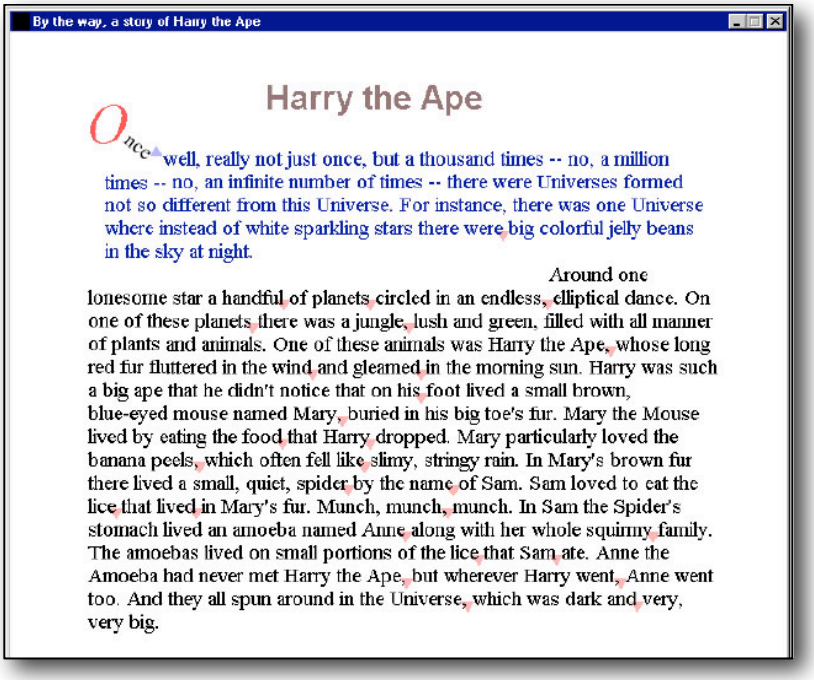


Figure 4.7: The fluid reader displaying 'Harry the ape' - a fluid hypertext narrative. The red icons represent visual cues which may be selected to introduce narrative digressions (depicted in blue).

Further applications of fluid document technology included spreadsheets (Igarashi *et al.* 1998) and a museum piece (Gold *et al.* 2000) in an exhibition entitled "eXperiments in the Future of Reading". The research also extended into the area of authoring and rendering fluid hypertext narratives such as 'Harry the Ape' (Zellweger *et al.* 2002) (See Figure *4.7*).

4.1.2.5 Evaluation

Fluid document technology was evaluated via an observational study focusing on the impact of the fluid techniques on reading and hypertext browsing behavior and effectiveness (Zellweger *et al.* 2000). The aim of the study was to determine whether the topological changes required to support fluid documents disrupted reading activity and how users interacted with and reacted to glosses and the techniques used to display them.

The results of the study indicated that gloss placement was important; placing glosses close to their source anchor improved reading efficiency while placing glosses outside of the primary text reduced negative reactions to typographic adjustments. In terms of interaction, the results indicated that lightweight interaction techniques were prone to inadvertent invocation of glosses and some readers reacted negatively to the use of animation. A number of interesting reading styles also emerged. Frozen glosses were used by subjects as reminders, to compare glosses and to search the hypertext in a breadth-first order.

4.2 Inline exploration and programmer disorientation

The approach to information exploration pioneered by the Guide hypertext system and the fluid document project is based on a fundamental principle of progressively incorporating related or secondary information directly into the context of a focal document or information display. This general concept is referred to as inline exploration, which is contrary to the more traditional mechanism of explicitly navigating between

discrete displays of information - the basic exploration mechanism prevalent in modern IDEs.

There exists a number of fundamental properties of inline exploration which are significant in terms of addressing programmer disorientation, particularly when exploring source code in an IDE setting. These properties are:

- Preservation of navigation history and context
- Elimination of cognitive disruption associated with explicit navigational transitions
- Support for simultaneous presentation of related information
- Support for exploratory digressions

4.2.1 Preservation of navigation history and context

A significant factor associated with the incidence of programmer disorientation in modern IDEs is a lack of navigation history and context during source code exploration activities (De Alwis & Murphy 2006; Janzen & De Volder 2003). As a programmer navigates from one program element to another in a system of source code, there is no explicit representation of navigation history in the interface, i.e. the path or sequence of previous source code elements or locations leading to the current program element. This context is important for reminding the programmer of their intent (Storey *et al*. 1999), supporting the development of a conceptual model and remaining oriented in the information space (Kim & Hirtle 1995;Thüring *et al*. 1995).

The process of inline exploration results in an explicit and interactive manifestation of navigation history in terms of the explored information itself. As a user progressively introduces related information into a focal display both the original display context as well as any introduced information fragments are visibly maintained in a sequential order. This representation of navigation context supports reflection, analysis and orientation in an information space.

A representation of navigation context may be sufficient to alleviate disorientation during intricate navigation activities in code space where a programmer will often lose track of their intent due to distraction associated with context loss, interface adjustment or external factors, particularly considering the dense, fragmented and complex topology of source code.

4.2.2 Elimination of cognitive disruption associated with explicit navigational transitions

Due to the keyhole display and lack of visual momentum in modern IDEs, source code is generally explored as a sequence of perceptually independent displays. The cognitive experience of traversing from one information context to another can be disruptive, particularly if the programmer follows a navigable link only to realize that the destination is not relevant to their current exploration goals. The programmer must leave the source context, enter the destination context, determine its relevance which may require secondary navigation steps, and then potentially re-enter the original source context and reorient to their original location. This process, which is continually repeated over the

course of a source code exploration task is cognitively draining and a significant factor related to disorientation problems (Conklin 1987; Foss 1989a; Foss 1989b; Zellweger *et al.* 1998). Using inline exploration a user can examine related information without the need to navigate out of the existing context. Related information may be introduced inline with existing information in a fluid and non distracting manner. This means that the cognitive disruption associated with explicit navigation is avoided thus potentially reducing the general tendency towards disorientation during exploration tasks.

4.2.3 Support for simultaneous presentation of related information

As discussed in chapter 3, a distinct property of source code is a high level of fragmentation. The implementation of coherent portions of system implementation is generally decomposed into a set of discrete source code elements which are spread across a number of source code documents in a physically disjoint and potentially complex manner. The typical manifestation of this phenomenon is ‘control flow scatter’ where the control flow associated with a program operation is fragmented across a number of disjoint source code elements (Chu-Carrol *et al.* 2003). The programmer comprehends this fragmented information via a keyhole display, essentially examining each piece of the implementation puzzle in isolation and then synthesizing an overall conceptual model from the various ‘glances’ at the code space.

The exploration of fragmented source code as a sequence of isolated displays is both cognitively draining and results in problems developing an accurate conceptual model (Thüring *et al.* 1995). This situation often results in the programmer thrashing

between individual displays in order to gain a sufficient overview, an activity which is indicative of disorientation and requires the programmer to concentrate on extraneous interface manipulation activities which may result in distraction and disorientation problems.

Inline exploration provides a framework for examining fragmented information in a single consistent display. Nested inline introduction means that a user can interactively introduce related portions of information into a single display in a controlled and selective manner thus providing the ability to achieve a semantically consistent and coherent overview. It is the proposition of this thesis that this ability would be particularly advantageous in a source code setting and would supersede the need to thrash between related displays in various situations.

4.2.4 Support for the pursuit of exploratory digressions

A factor in the incidence of disorientation in modern IDEs is a lack of support for the pursuit of exploratory digressions (De Alwis & Murphy 06). When exploring source code, a programmer is continually faced with the need to pursue and evaluate navigational branches in the code space. In the absence of a visible representation of navigation history, pursuit of exploratory digressions can result in the embedded digression problem (Foss 1988a) whereby a programmer loses track of their intent and fails to return from a particular digression or fails to pursue a planned digression.

Inline exploration provides support for the pursuit of small scale digressions in context. A user can explore into the information space associated with the focal document

or information display without leaving the original context. If the digression proves to be unrelated to the current exploration goals, the navigation path can be easily dismissed leaving the user in the original context and requiring no further backtracking and reorientation. Furthermore, fragments of information can potentially be introduced into a focal display and left open as a reminder of a planned digression. This behaviour has been observed during the evaluation of the fluid document technology (Zellweger *et al*. 2000).

4.3 Inline exploration and program comprehension

Programmers explore source code in order to comprehend a particular part or feature of a software system, generally in the context of a given development or maintenance task (Singer *et al.* 1997). The programmer's mental model refers to their current knowledge of the system, such as how particular operations are carried out and how components interact to facilitate observable system behaviour. A cognitive model describes the cognitive processes and information structures used by the programmer to develop and refine their mental model. This process is based on existing knowledge, the code itself and available documentation (Von Maryhauser & Vans 1995). A number of cognitive models have been proposed to describe how programmers comprehend code. In this section we discuss the core comprehension models and how they may be facilitated or affected by the inline exploration technique.

4.3.1 Bottom-up comprehension

Bottom-up theories of program comprehension maintain that programmers develop a mental model by reading source code and chunking individual statements, fragments of code and program elements into higher level abstractions based on their structure and relationships. These abstractions are eventually synthesized into an overall comprehension of the system, or part thereof (Schneiderman & Mayer 1979, Schneiderman 1980).

Bottom up is the most relevant comprehension theory in the context of the inline exploration discussion. Inline exploration facilitates the examination of multiple related source code elements in a single display, and also visually renders the relationships between such elements allowing the programmer to gain an overview. This feature has the potential to improve a programmers ability to draw high level abstractions from fragments of code scattered within or across a number of source files. Furthermore, Pennington (1987) observed that programmers initially develop a control flow abstraction of the program, capturing the sequence of operations. This is referred to as the program model. Again the development of the program model may be supported by the ability to examine related source code elements from non contiguous locations in the code space. The program model is subsequently combined with the situation model, based on domain knowledge, data flow and function abstractions to develop a mental model of a system.

4.3.2 Top-down comprehension

Brooks (1983) describes that programmers understand systems in a top down manner by mapping knowledge about the application domain to patterns in the code. The process is based on a hierarchal system of hypotheses which are gradually refined or rejected based heavily on the existence of beacons within the code. A beacon is a pattern or set of features within the code which indicate the existence of hypothesized structure or operations. Beacons are significant in terms of inline exploration. The ability to overview code structure larger than that of a single source code display may make it easier for programmers to identify beacons which span across a number of source code locations and documents.

Soloway & Ehrlich (1984) observed that programmers build a top down model based on the existence of programming plans and rules of programming discourse within the code. Programming plans are fragments of code which indicate typical programming scenarios and rules of programming discourse reflect convention such as standards and common implementations. Again inline exploration should help programmers to identify such constructs in the event that they span multiple source code locations.

4.3.3 A note about the mixed model

Letovsky (1986) considers programmers as ‘opportunistic processors’ capable of switching between a top down and bottom up comprehension model as needed. Von Maryhauser & Vans (1995) observed programmers frequently switching between comprehension models.

4.4 Envisioning Inline source code exploration

Inline source code exploration is fundamentally limited to being a ‘within source-code’ exploration technique. This means that the approach is applicable only in a situation where the programmer is navigating from within the source code editor presentation itself. Essentially the editor display represents the context into which related source code fragments may be introduced by the programmer. The currently predominant technology in this exploration space is hypertext style exploration via program cross-references - where the programmer clicks on simulated links related to program references in the source code display to navigate to associated program declarations. The execution of searches based on references and declarations selected in the editor display, the use of pop-up source code displays and the ‘declaration view’ are also relevant technologies.

The high level model of inline source code exploration in the IDE is that the programmer interacts with program references presented in the visible source code editor display to achieve the inline introduction of corresponding source code declarations. Furthermore, from within introduced declarations, its should be possible for the programmer to introduce further nested declarations where applicable, thus achieving the full potential of the inline exploration experience.

Chapter 5

The Fluid Source Code Editor

To realize, and conceptually explore, the notion of inline source code exploration an inline exploration interface for source code entitled the ‘fluid source code editor’ (Sourceforge 2009) was developed. The fluid editor is an open source extension of the Eclipse IDE which facilitates the inline exploration of Java source code. This chapter describes the design and implementation of the fluid editor.

5.1 Preliminaries

The fluid editor is implemented as a series of plug-ins for the Eclipse IDE. A plug-in is the fundamental unit of extension in the Eclipse platform. Each plug-in, which is essentially a jar file, contains a *plugin.xml* file which targets particular platform extension points with custom implementation. The primary plug-in making up the the fluid editor extension, *org.eclipse.fluid.ui*, contributes a new editor type to the Eclipse platform

associated with Java source code documents (*.java). The fluid editor plug-in targets the *org.eclipse.ui.editors* extension point with a custom editor class which extends the standard Java source code editor provided by the JDT.

After installation of the fluid editor plug-in set into Eclipse, Java source code documents are opened, by default, in a fluid editor instance as opposed to the standard Java editor provided by the Eclipse JDT.

5.2 System overview

The fluid editor facilities the inline exploration of source code by means of fluid annotations embedded in the source code editor presentation, and the inline introduction of source code declarations and other information types.

When a Java source code document is opened in a fluid editor instance, a fluid annotation is embedded alongside each source code reference in the editor presentation. A fluid annotation is essentially a lightweight marker or visual cue embedded in the source code which indicates the existence of a related source code declaration to the programmer. The document into which fluid annotations are embedded is referred to as the focal, or primary, source code document, as it is the document into which inline introduction occurs and is the focus of the programmer during inline exploration activities.

When a fluid annotation is activated by the programmer, the corresponding source code declaration is dynamically introduced, inline, into the primary source code

document. Introduced source code declarations may also contain fluid annotations which can be activated to achieve nested introduction. Essentially, a source code declaration may be introduced into the context of an existing inline declaration, thus facilitating progressive nested inline exploration.

The fluid editor supports the annotation and introduction of a comprehensive set of Java language declarations. The supported set of reference/declaration tuples are listed in Table *5.1*.

Reference to declaration mapping in the fluid source code editor	
Reference type	Introduced declaration
Method invocation	Method declaration
Constructor invocation	Constructor declaration
Super constructor invocation	Supper constructor declaration
Super method invocation	Super method declaration
Class instance creation	Class constructor declaration
Type reference	Type declaration (without imports)
Field reference	Field declaration
Local variable reference	Variable declaration

Table 5.1: Reference type to inline declaration mapping used in the fluid source code editor.

5.2.1 Fluid annotations

A fluid annotation is an unobtrusive visual cue embedded in the source code presentation which indicates the existence of a corresponding source code declaration. The default appearance of a fluid annotation is a single character underline (See Figure *5.1*). The minimal profile is designed to avoid eroding the readability and editability of the underlying source code document. Fluid annotations may also be 'maximized' to a more visible state - in which case they appear as transparent boxes (See Figure *5.2*).

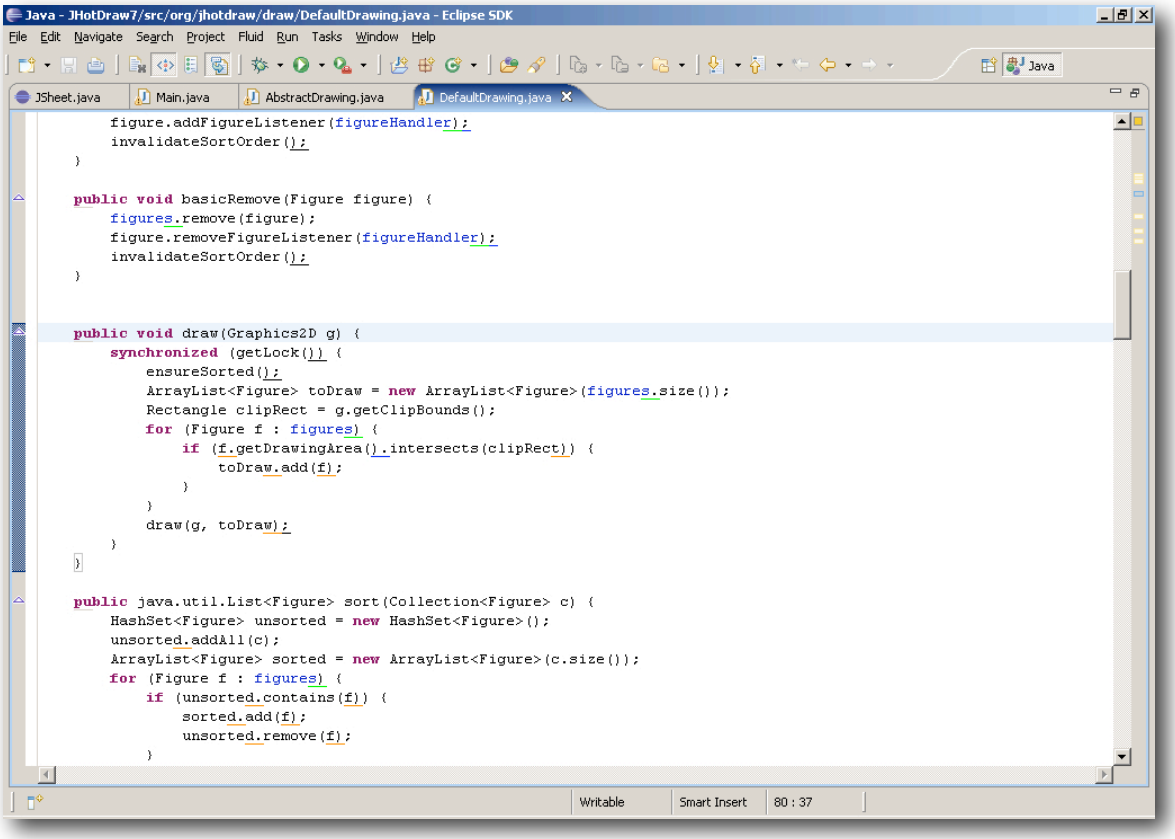


Figure 5.1: Fluid annotations embedded in the focal source code editor presentation. Colour indicates the type of reference. In this instance black refers to method references, green to field reference, orange to local variable declarations and blue to abstract method references.

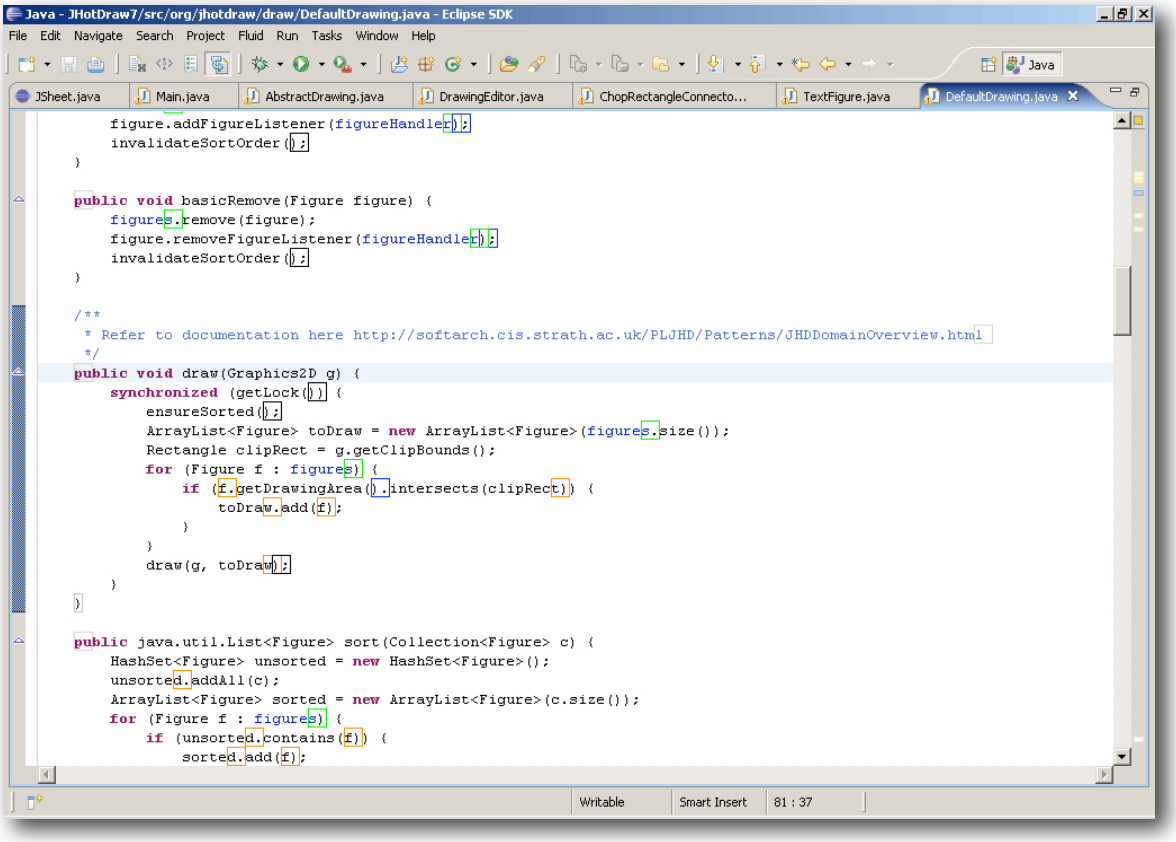


Figure 5.2: Fluid annotations in their 'maximized' state.

Fluid annotations are designed with source code editing in mind. Using the editor caret, the programmer can easily edit around and directly upon fluid annotations during programming activities. The fluid editor takes care of repositioning annotations in response to edits, removal of annotations that are no longer valid and insertion of new annotations as the programmer adds additional source code to the document. The updating of annotations occurs in real time as the programmer edits the source code.

Fluid annotations are colour coded to indicate reference type. This allows the programmer to distinguish individual annotations when a cluster of heterogeneously typed annotations are located in close proximity - which is often the case with densely

written source code. The fluid editor provides a preferences page allowing the programmer to change the mapping between reference type and annotation colour.

When the programmer moves the mouse within a fixed proximity (a one character bounding box) of a fluid annotation it dynamically transforms into an interactive widget. Each fluid annotation has two states, a visually minimal state in which the annotation acts as a visual cue, and a widget state in which the annotation may be interacted with by the programmer to initiate introduction of the corresponding source code declaration. State transition is achieved by rolling the mouse pointer over the annotation (See Figure *5.3*). Clicking the widget associated with a fluid annotation causes the corresponding source code declaration to be introduced into the source code presentation. The widget associated with a fluid annotation also has two states, an 'expandable' state and a 'collapsible' state. The widget is in an expandable state when the associated source code declaration has yet to introduced and a collapsible state when the associated declaration is introduced and may subsequently be collapsed.

Fluid annotations are located, by default, on the last character of the associated source code word for variable references, and on the last enclosing parameter bracket for method references. The fluid editor also provides a preference setting to facilitate the placement of fluid annotations on the last character of a method name if the user finds this technique more agreeable. The placement of fluid annotations is designed to allow the user to easily match a given annotation to its associated source code element.

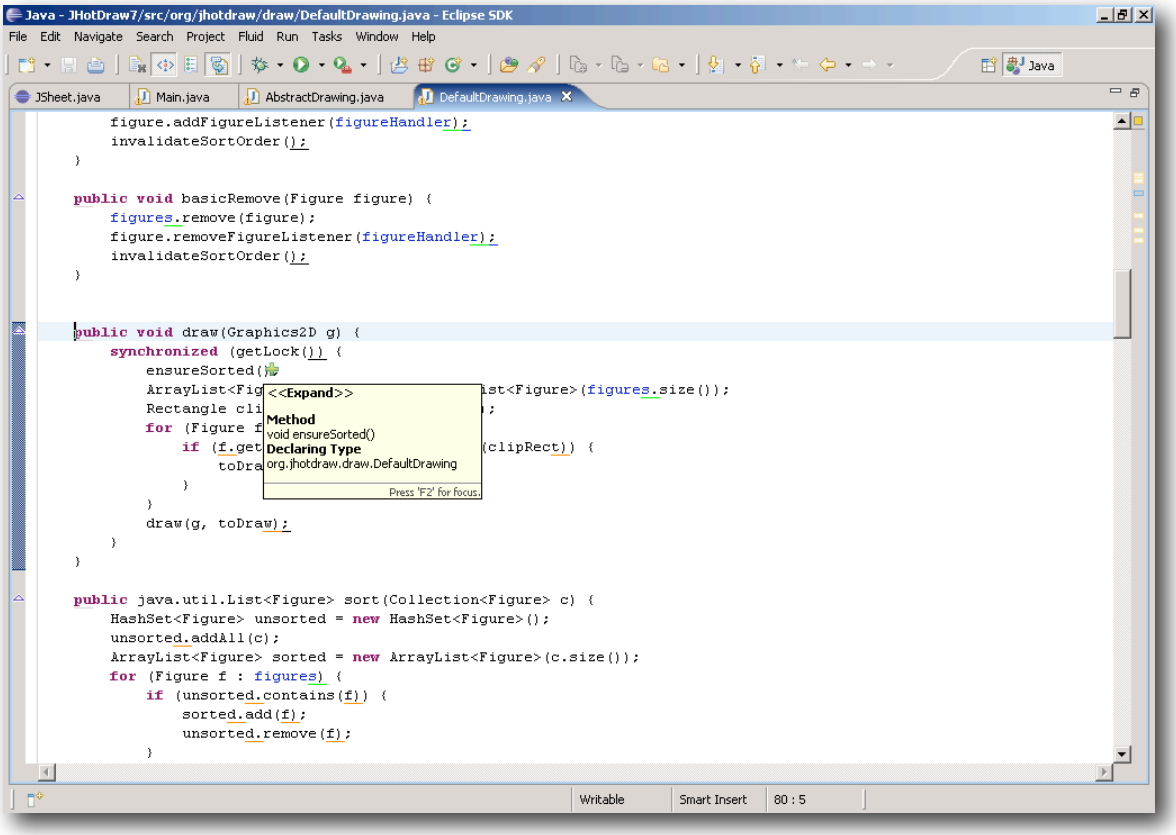


Figure 5.3: When the mouse cursor is positioned within the bounding box of a fluid annotation is transforms into an expandable widget - in this instance a green plus icon affording addition or expansion. After a number of seconds a information box will also appear describing the potential introduction.

5.2.2 Inline introduction

Inline introduction refers to the introduction of a source code declaration (or other units of information) into the context of the focal source code document. The fluid editor uses an inter-line introduction technique inspired by Zellweger *et al.* (2000). See Chapter 4, Section 1.2.2. The source code document is split horizontally at the line succeeding the fluid annotation and the source code declaration is then inserted as an indented code

block (See Figures *5.4* & *5.5*). Early iterations of the fluid editor used a line splitting introduction technique in which the code was split at the character succeeding the visual cue. However, by splitting lines of code in this manner the source code became difficult to read particularly with multiple introductions originating from a single line of code. The current 'after line' technique was developed to conserve the structural integrity of the line of code associated with the fluid annotation(s) and thus increase the overall readability and coherence of the source code display.

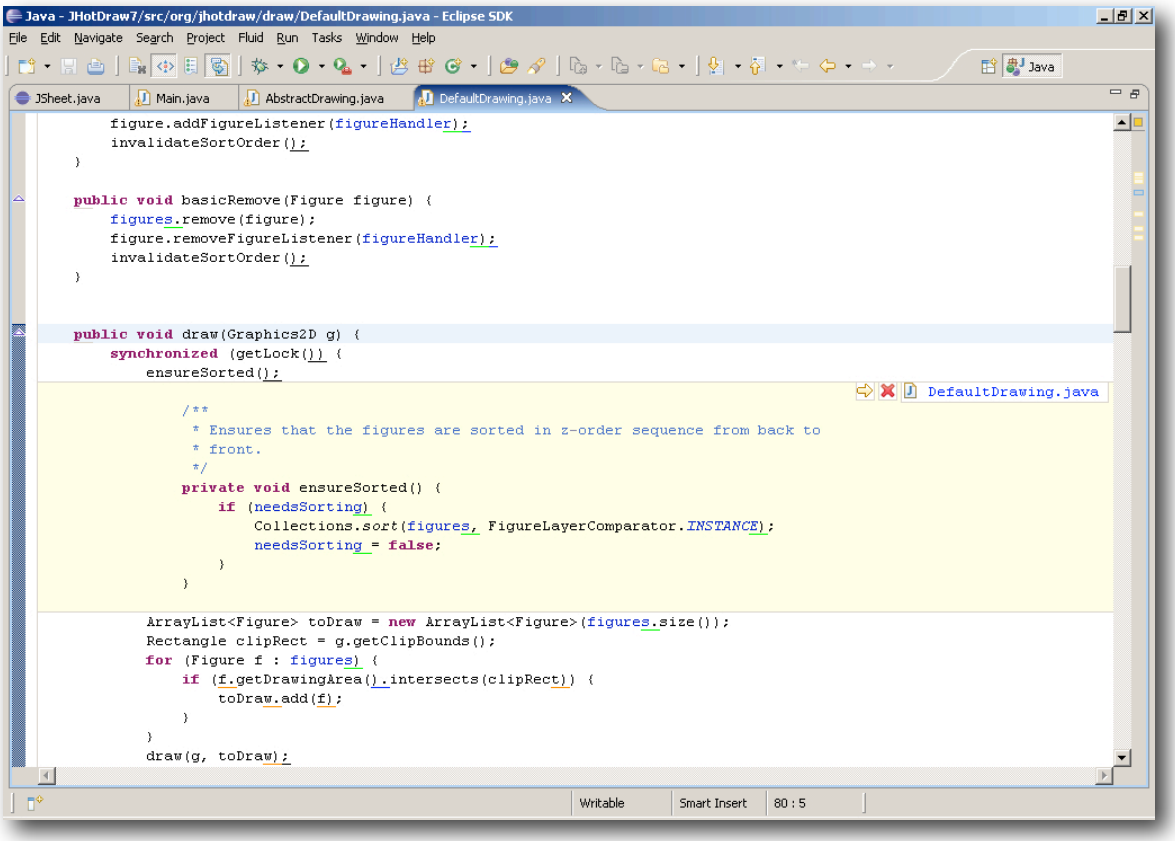


Figure 5.4: An introduced method declaration. The background colour is distinct and the inline declaration has a border to allow the user to easily identify between introduced and native code.

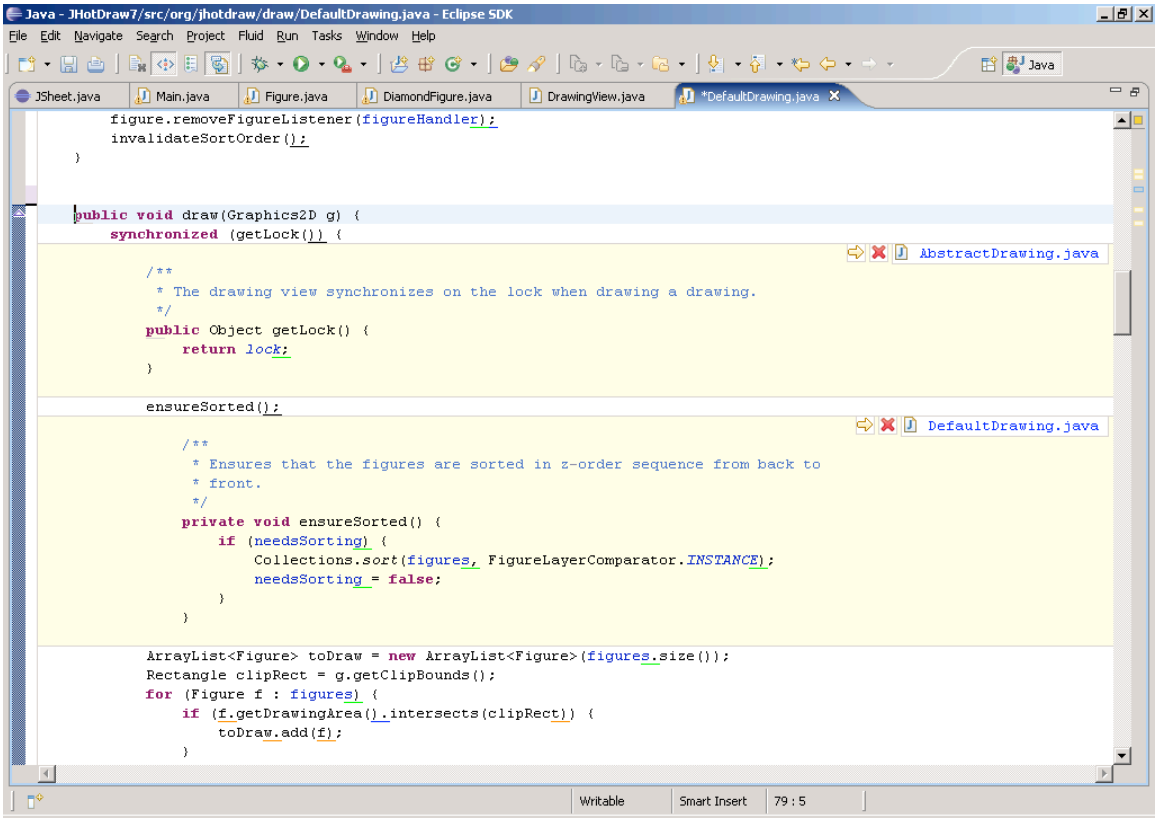


Figure 5.5: Introduction of a second source code declaration.

When designing the fluid source code editor a variety of introduction techniques were considered, such as interline expansion, margin callout and fluid overlay. These design alternatives were primarily inspired by the fluid UI techniques developed as part of the fluid document project (Zellweger *et al*. 1998) and partially the Guide 'in-situ' mechanism (Brown 1989). See Chapter 4, Section 1.2.2 and Chapter 4, Section 1.1.2. After a significant amount of technical and usability oriented experimentation, an interline introduction technique was chosen. The following sub-sections expand on this pivotal design decision.

5.2.2.1 Margin callout

The margin callout technique is concerned with the introduction of related information into the margins either side of a digital document. See Chapter 4, Section 1.2.2. Source code editors generally do not support margins (however a ‘gutter’ mechanism is supported in many IDEs). As such initial experimentation was carried out based on the addition of margins into which source code declarations could be introduced. However, a number of usability issues quickly became clear. The first issue was margin width and its effect of on the primary document context. The margin width necessary to introduce a source code declaration, without the need to scroll or artificially reduce the size of the introduction, infringed upon the readability of the original document context. Furthermore, the margin callout technique could not support the level of rich nested exploration that envisioned for the fluid source code editor. For these reasons the margin callout technique was not pursued beyond a number of basic experiments.

5.2.2.2 Fluid overlay

The fluid overlay technique (See Chapter 4, Section 1.2.2) was problematic from a technical standpoint due to a fundamental lack of support for changing the transparency of user interface controls in the Standard Widget Toolkit (SWT) (Eclipse 2009g) - the user interface framework used in the Eclipse IDE. Essentially the SWT provides a standard Java based user interface API which is mapped, via the Java native interface (JNI), to the underlying native user interface API on platforms such as Microsoft

Windows, Apple Mac and Linux. Because the SWT needs to cater for a wide variety of native interface frameworks, each with varying degrees of functionally, the interface tends to revert to a lowest common denominator in terms of available functionality. A casualty of this situation is the ability to change the transparency of interface controls. Without the ability to create transparent interface controls the application of the fluid overlay introduction technique was unfeasible.

5.2.3 Inline source code declarations

Inline source code declarations are read only copies of native source code declarations. The fluid editor supports the introduction of methods, type and variable declarations (both local variables and fields) (See Table *5.1* for a full listing of supported introduction types).

The user can copy the code contained within an inline declaration but it cannot be edited. While editing of the source code contained in an inline declaration is technically feasible with the current implementation, the approach was not pursued as the primary focus was on contextual exploration as opposed to out of context editing, which in itself is a significant research question. Other than being read-only, inline source code is fully consistent with native code. It is syntactically and semantically highlighted and supports standard caret and line highlighting behaviour. The user can also navigate the source code contained in an inline declaration using the standard out of line hypertext exploration model, run searches via the context menu and initiate tool tips.

Inline source code declarations are differentiated from native source code by a border and a coloured background. For instance, the default background colour for a method declaration is a soft yellow with a grey border. In order to keep the user oriented during inline source code browsing, inline declarations are labelled with the name of their native source code document. A toolbar is also provided which allows the user to collapse a given declaration or navigate to the associated source code declaration in its native context. Explicit navigation is also supported using a ‘shift select’ combination. If the shift key is held and the mouse moved within the bounds of an inline declaration, it becomes a navigable link to its native source code declaration. To indicate this feature the cursor transforms into a hand pointer and the background of the inline declaration turns to a deep yellow indicating its highlighted status (See Figure *5.6*). When clicked the target declaration is opened in a new source code editor and presented to the user.

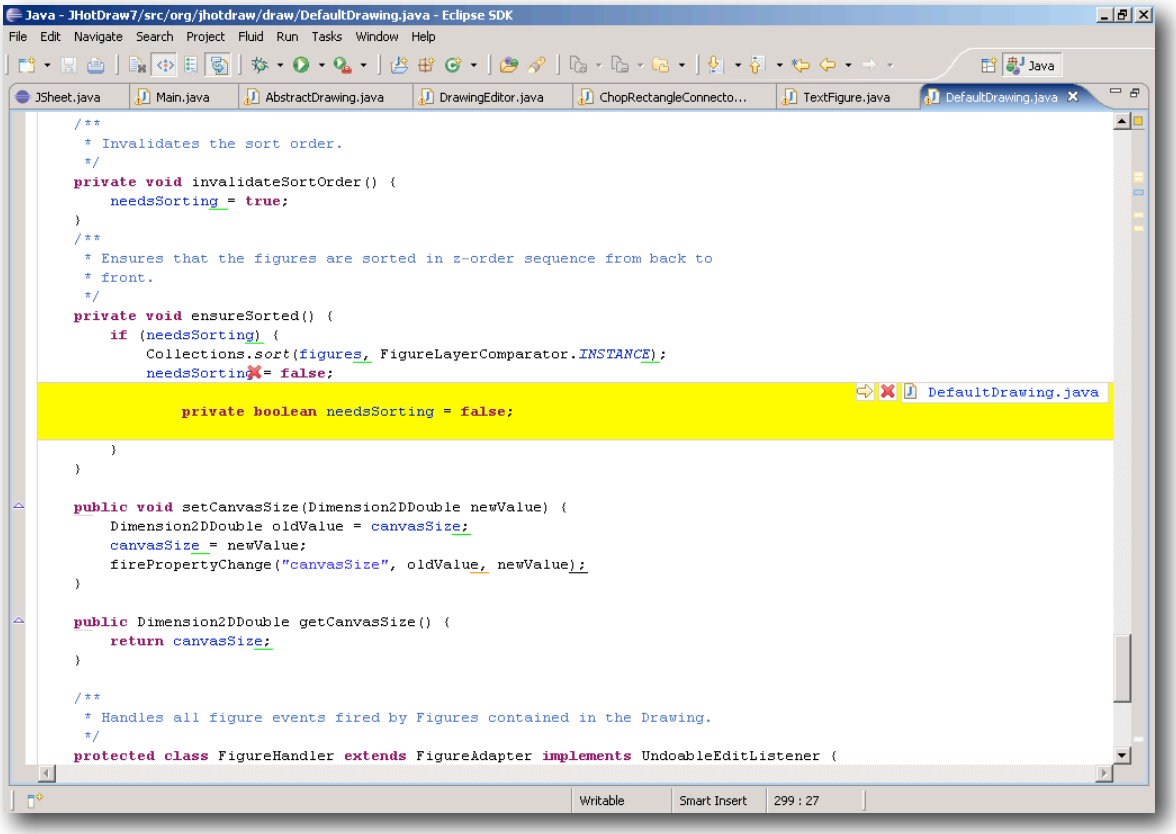


Figure 5.6: Explicit navigation using an inline declaration. The user has held the shift key and moved the mouse pointer over this field declaration causing it to become a navigable link. Clicking on an inline declaration when it is in a navigable state will open the corresponding native source code declaration in a new editor display.

An introduced inline declaration may be ‘collapsed’ or removed from the source code display by re-clicking on its associated fluid annotation (which is in a ‘collapsible’ state) or clicking on the close button contained in the inline toolbar. When the mouse pointer is positioned within the bounds of an inline declaration, the associated fluid annotation is displayed in the more distinct widget state. This allows the user to easily identify both the source code reference and the fluid annotation associated with the inline declaration.

5.2.4 Nested introduction

It is common for source code contained in an inline declaration to contain references to other source code declarations. To handle this scenario and fully realize the inline exploration concept the fluid editor supports the nested introduction of source code declarations (See Chapter 4, Section 1.1.2 and Chapter 4, Section 1.2.3 for discussion related to nesting in an inline context). This means that an inline declaration may be introduced into the context of an existing inline declaration (See Figures *5.7* & *5.8*).

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

An entire 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.

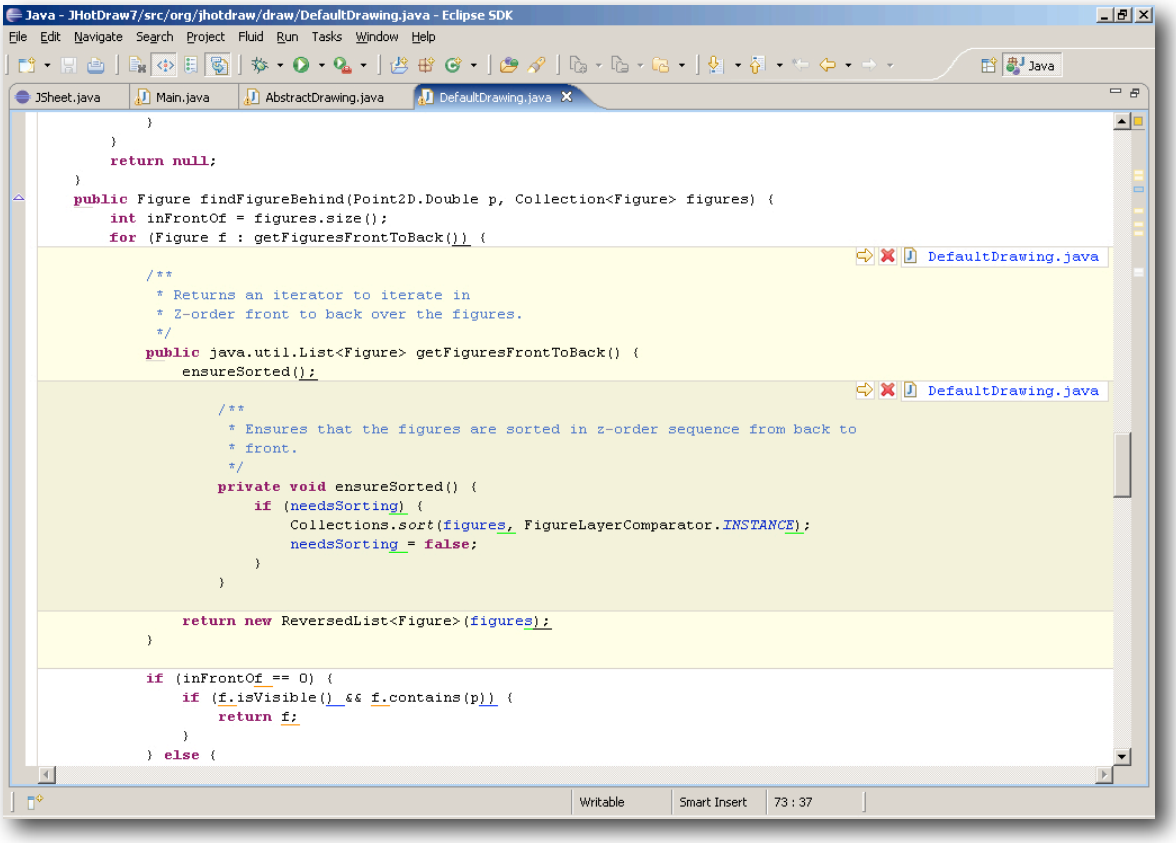


Figure 5.7: Nested inline introduction. In this instance one method declaration has been introduced into the context of an existing inline declaration. Nested declarations are differentiated via a colour model, in this case shading based on introduction depth.

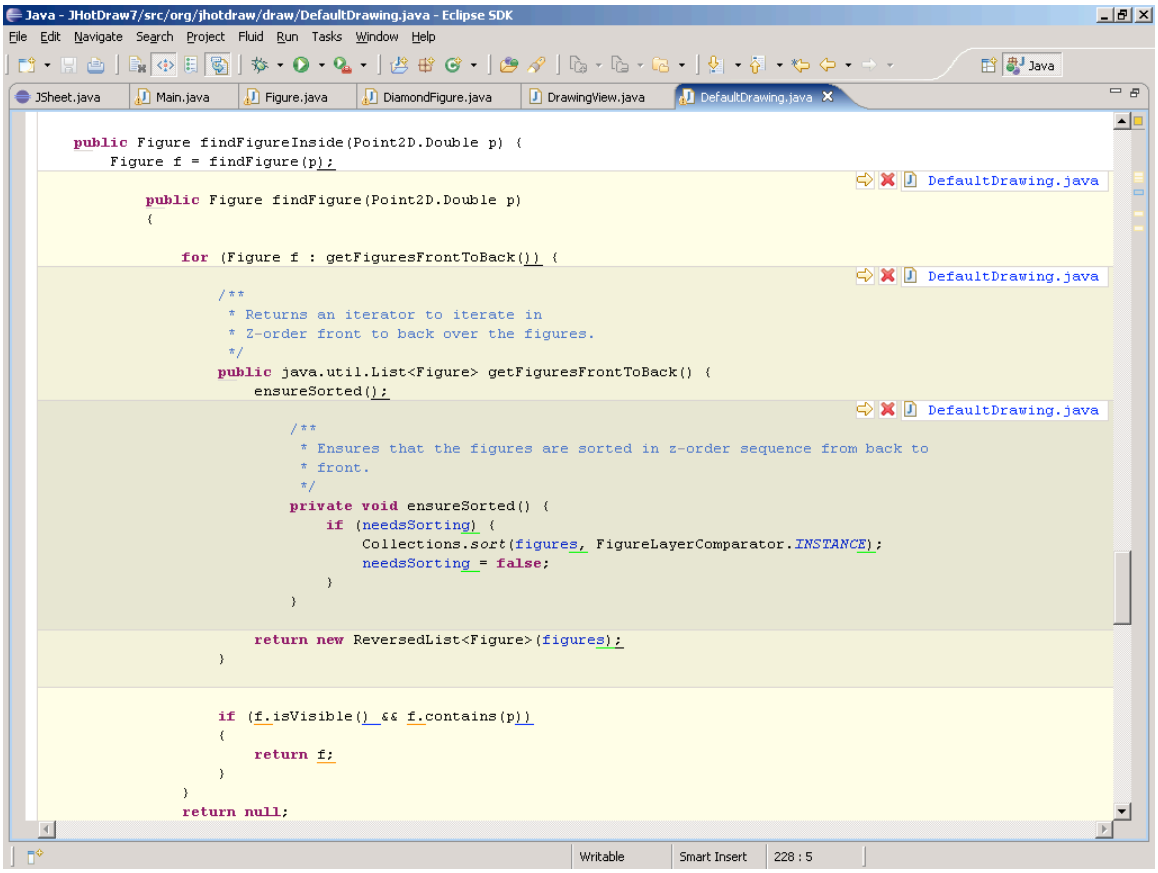


Figure 5.8: Tree levels of inline nesting and the corresponding depth shading.

5.2.5 Search results

In addition to the introduction of basic source code declarations, the fluid editor also supports the introduction of search results as a core feature. The programmer can execute a ‘pre-canned’ search from within the source code presentation by activating particular fluid annotations. The results of the search are then presented inline and the programmer can explore the set of matches using the inline exploration mechanism (See Figures *5.9* & *5.10*).

The execution and introduction of search results in an inline manner offers a significantly more contextual and less disorienting search experience than the existing approach prevalent in modern IDEs. In the traditional IDE environment search results are displayed in an external results view, and when the programmer navigates to a particular result the original context of the search is replaced and lost (De Alwis & Murphy 2006). Loss of the original search context is significant as it represents an important reminder of the intent underpinning the search. When it is no longer visible the programmer is more prone to disorientation. Furthermore, due to the keyhole property it can be cognitively demanding to compare and contrast a set of search results. The programmer may need to thrash back and forth between particular source code locations which places a strain on working memory and requires greater concentration on interface adjustment activities. On the other hand, using the inline search mechanism the original context of the search is maintained and the programmer can selectively compare and contrast individual results in a single contextual coherent display.

The fluid editor supports the introduction of search results in a number of scenarios most of which are related to polymorphism - in which there may be many declarations associated with a particular source code reference (See Table *5.2* for a list of supported references). The inline search support provided by the fluid editor is not intended to be complete by any means but rather sufficient enough to evaluate the feasibility and usability of the concept. It is envisioned that inline search results would be a far broader feature, encompassing a much richer set of search functionality.

Inline introduction of search results	
Source code element	Results
Method invocation on an interface type	All implementing declarations in the workspace
Method invocation on an abstract type	All implementing declarations in the workspace
Interface type declaration	All implementing declarations in the workspace
Interface method declaration	All implementing declarations in the workspace
Abstract type declaration	All implementing declarations in the workspace
Abstract method declaration	All implementing declarations in the workspace

Table 5.2: A summary of inline search support in the fluid source code editor.

In order to keep the discussion of inline search results succinct, the introduction of a method invocation on particular interface type will be discussed. General behaviour may be extrapolated from this description.

Upon activation of the fluid annotation associated with the method, the fluid editor first runs a workspace wide search for all matching declarations using the Eclipse search APIs. Once the search is complete, an inline display is introduced containing the list of matching declarations. Each entry in the list contains an associated fluid annotation (See Figure *5.9*). When an annotation in the results list is expanded, the associated source code declaration is introduced (See Figure *5.10*). Each matching declaration can also be treated as a hyperlink facilitating explicit navigation to the native declaration. The introduction of a search result takes slightly longer than a standard declaration, due to the additional processing required, 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.

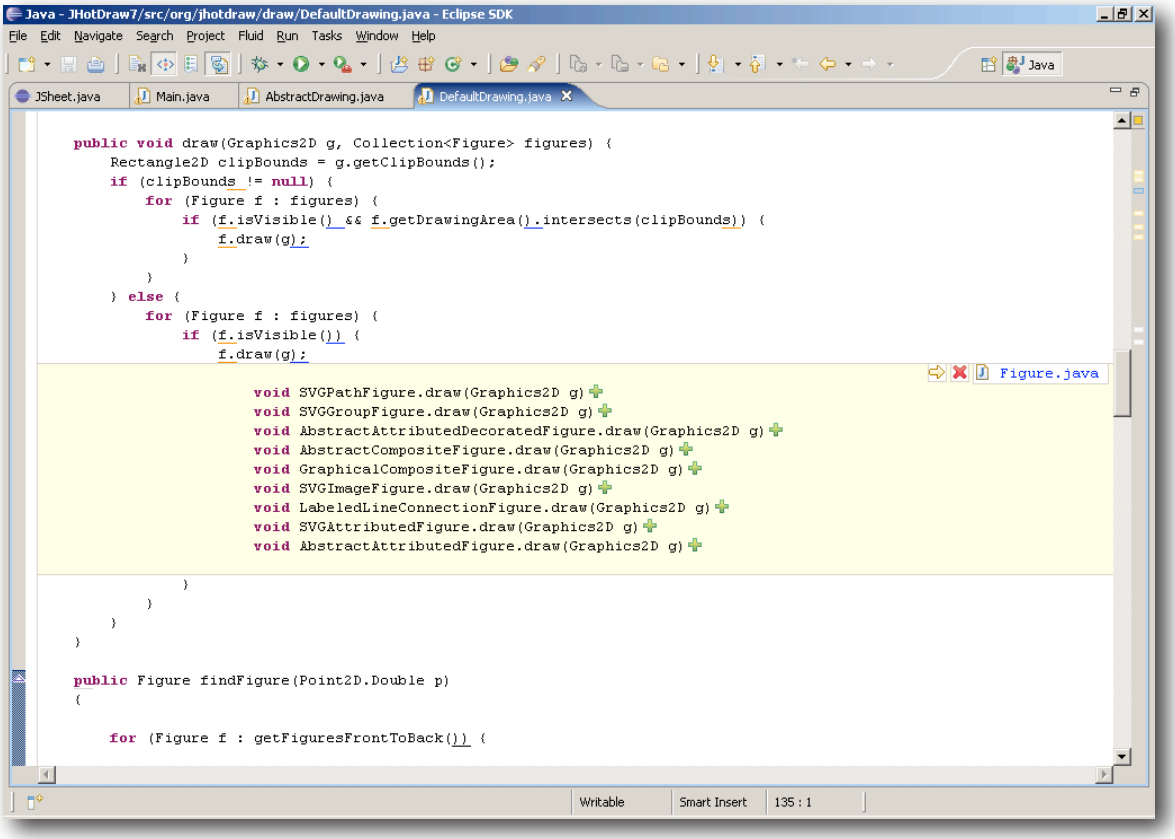


Figure 5.9: Inline introduction of search results - the declarations matching a particular abstract method invocation. The user can explore the results inline via the fluid annotation associated with each result.

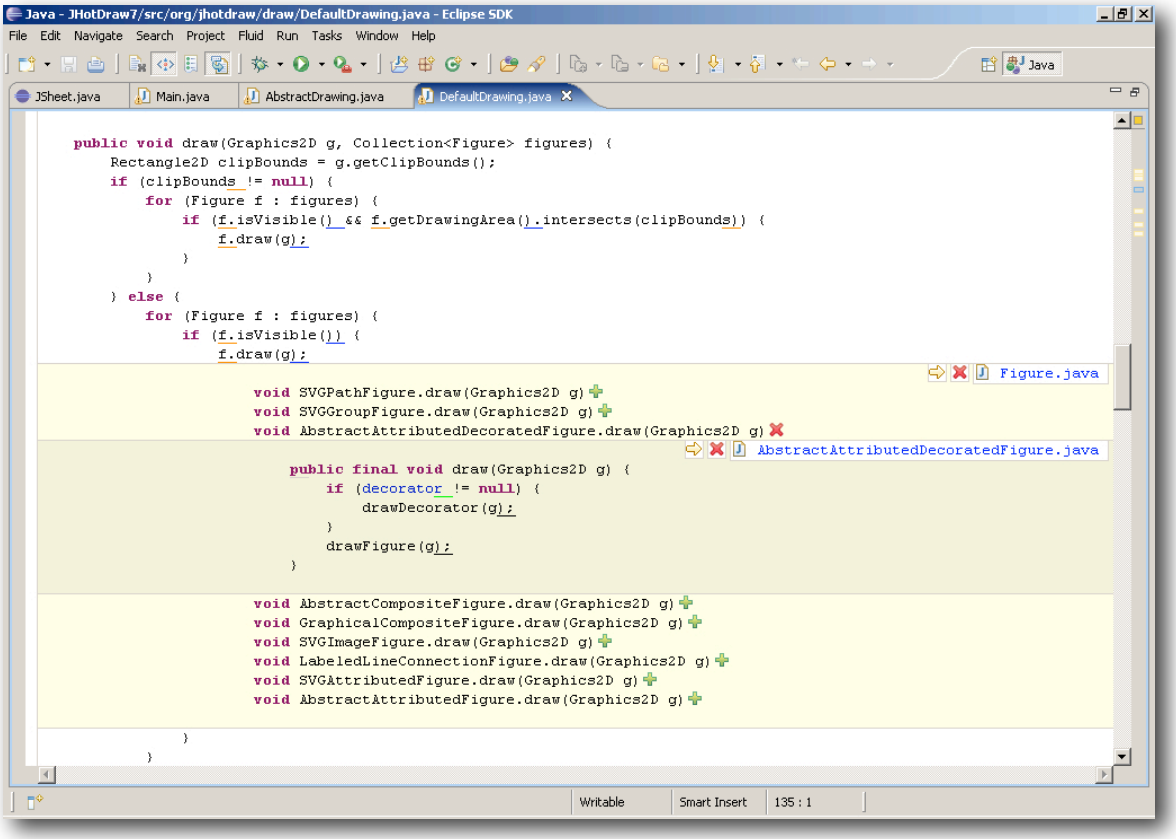


Figure 5.10: Exploring search results in an inline manner.

5.2.6 Inheritance relationships

In a Java programming environment, where inheritance is pervasive, a given method declaration will often ‘implement’ an abstract method declaration or alternatively override a super class method declaration. The fluid editor facilitates inline exploration in both circumstances even though there is no explicit reference on which to associate a fluid annotation.

When a method is deemed to implement or override a super class method, a fluid annotation is embedded at the first character in the method signature. The widget

associated with the fluid annotation is indicative of the inheritance relationship present, ‘implements’ relationships are represented as a white upturned arrow and overrides relationships are represented as a green upturned arrow as per JDT style standards.

When the fluid annotation is activated the appropriate method declaration is introduced above the existing method declaration (See Figure *5.11*). Using this feature the user can examine the structure of a type hierarchy from the point of view of a particular method all the way up to the abstract declaration within a single source code display.

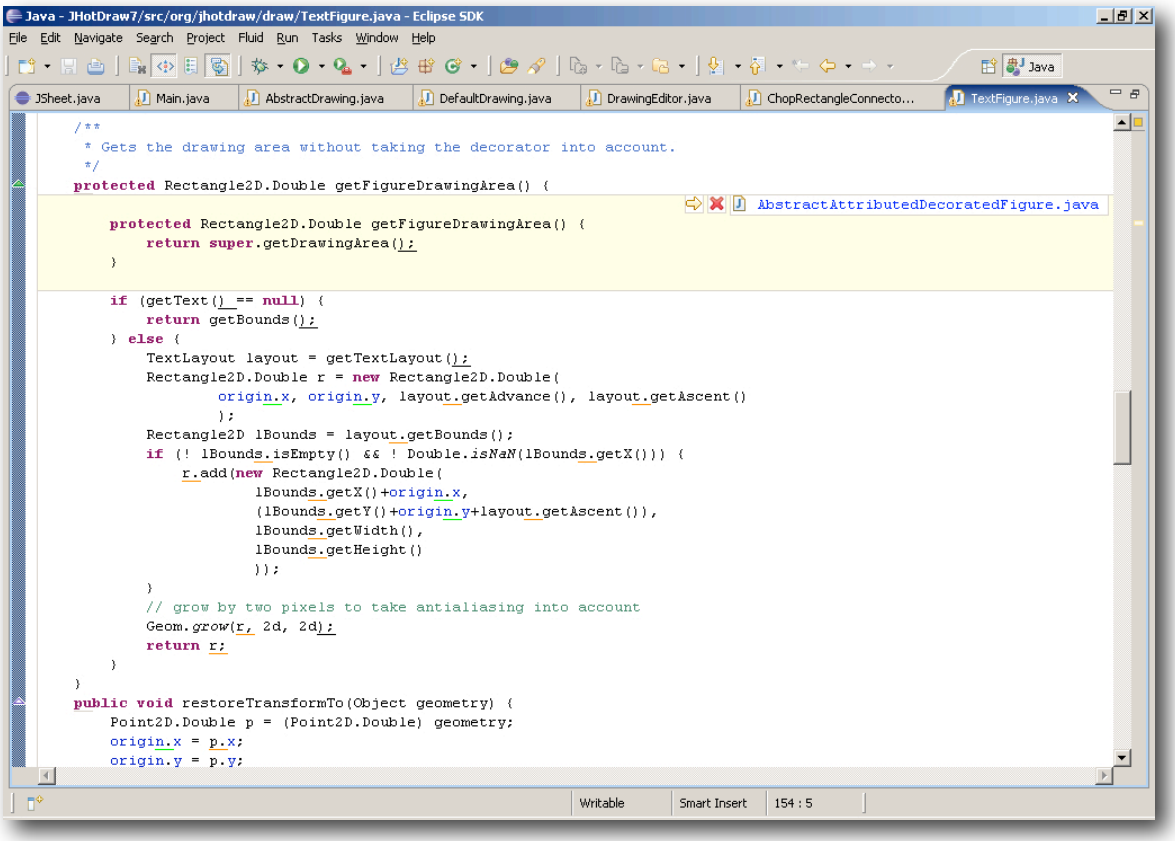


Figure 5.11: Introduction of overridden super-class method. The fluid annotation is located as the first character in the method signature.

5.2.7 Editing and reconciliation

The fluid editor is, by definition, an editor and thus supports source code editing in parallel with inline exploration activities. As a result the reconciliation of fluid annotations and inline source code declarations in response to editing activities is a significant design concern.

5.2.7.1 Reconciliation of fluid annotations

As the programmer edits source code using the fluid editor, fluid annotations are automatically inserted, removed and repositioned as required to maintain the integrity of the exploration environment. The repositioning of annotations occurs in real time while addition and removal of annotations occurs when the source code document is saved by the user. Addition and removal is delayed until saving as the procedure is processor intensive and would result in slight delays to editing actions if carried out in real time.

The addition of fluid annotations is typically carried out in response to the programmer inserting new code into the focal source code document. For instance, when adding a new method or variable reference. Removal of visual cues occurs in response to the deletion of code or the manual alteration of declaration signatures thus invalidating the link between a reference and an associated declaration. An example would be the alteration of a method signature which would invalidate any existing references to the old signature.

5.2.7.2 Reconciliation of Inline declarations

As mentioned previously, the source code contained within an inline declaration is a read only copy of a corresponding native source code declaration. This raises the question of the protocol used when a native declaration is edited or deleted and there exists inline copies in open fluid editor instances. If native declaration edits were simply ignored, the situation would invariably arise where an inline declaration would become ‘stale’ or inconsistent in relation to its target native declaration due to routine source code editing activities.

The fluid editor uses a simple but effective update notification mechanism to handle the editing of program declarations. When a native program element declaration is edited and subsequently saved, any inline copies of the edited declaration are checked for consistency. If the native declaration and the copy are deemed to be inconsistent, the inline declaration is flagged as such. Inconsistent inline declarations are indicated with a red outline border. When the mouse is hovered inside the inconsistent declaration a pop-up window also appears to indicate the problem. To synchronize the contents of the inline declaration with the newly edited native declaration the programmer simply removes and reintroduces the declaration using the associated fluid annotation, this action automatically refreshes the contents of the declaration.

Automatic updating of stale inline declarations was considered but proved problematic due to complications arising from nested declarations. An inline declaration may be ‘split’ arbitrarily by any number of inline child declarations. To implement an

automatic reconciliation strategy would involve significant complexity in terms of maintaining and managing the reintroduction of nested declarations.

In the case of deletion or the alteration of the signature of a native declaration, the fluid editor will automatically collapse all corresponding inline copies. The associated fluid annotation may then be deleted itself or updated if part of a re-factoring operation.

5.3 Additional features

The primary functionality of the fluid editor is the inline introduction of source code declarations and search results. However the fluid editor also supports the introduction of additional information such as web pages and images resources. The introduction of additional information artifacts such as dynamically computed program slices, life-cycle artifacts such as design diagrams and requirements was also experimented with using the framework. This section describes various experimental inline exploration facilities supported by the fluid editor.

5.3.1 URLs

Web accessible uniform resource locators (URLs) are often located in source code documents linking to licensing information and other documentation associated with the source code. The fluid editor supports the introduction of URLs via a lightweight inline web browser.

Any URL string encountered in the source code is annotated as standard with a fluid annotation (See Figure *5.12*). When the user clicks on the annotation, a web browser instance is introduced into the editor display and the target URL is opened (See Figure *5.13*). The web browser instance adopts a fixed height, specified in the fluid editor property pages, and scroll bars appear if the web content exceeds the available vertical space.

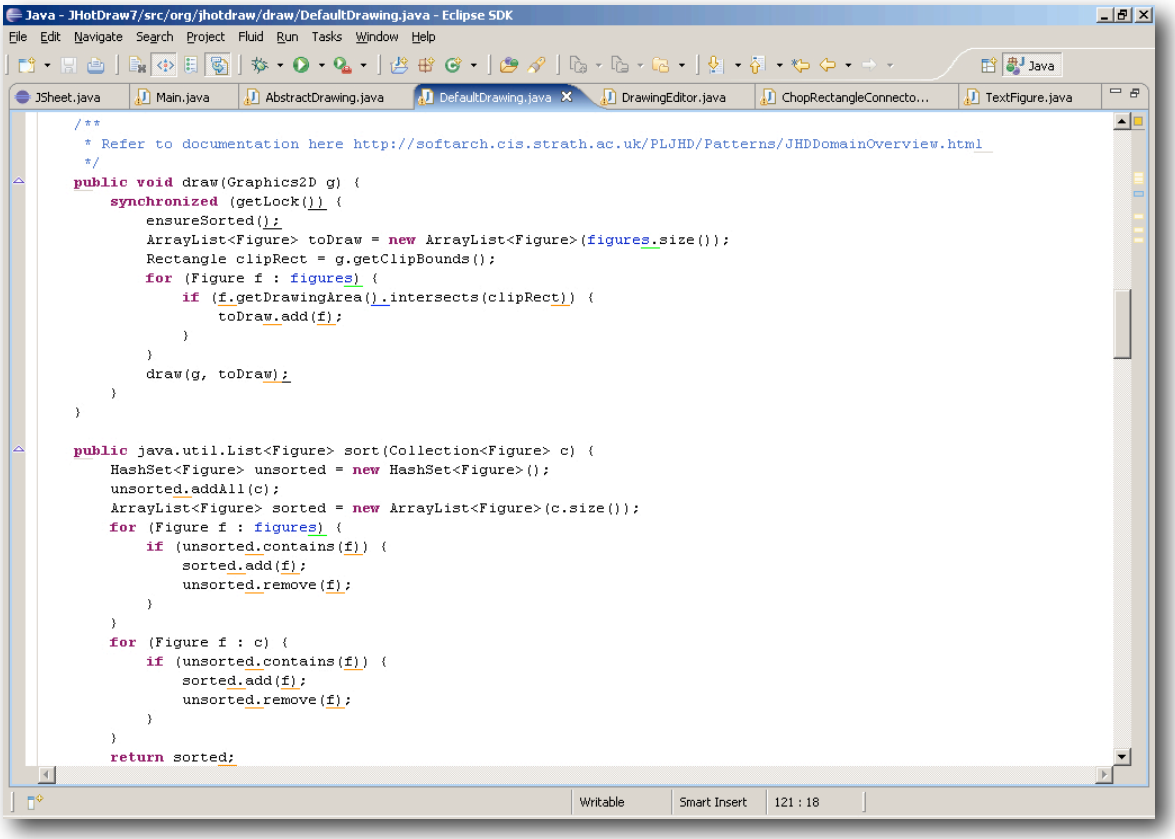


Figure 5.12: A url contained in a comment block is annotated with a fluid annotation.

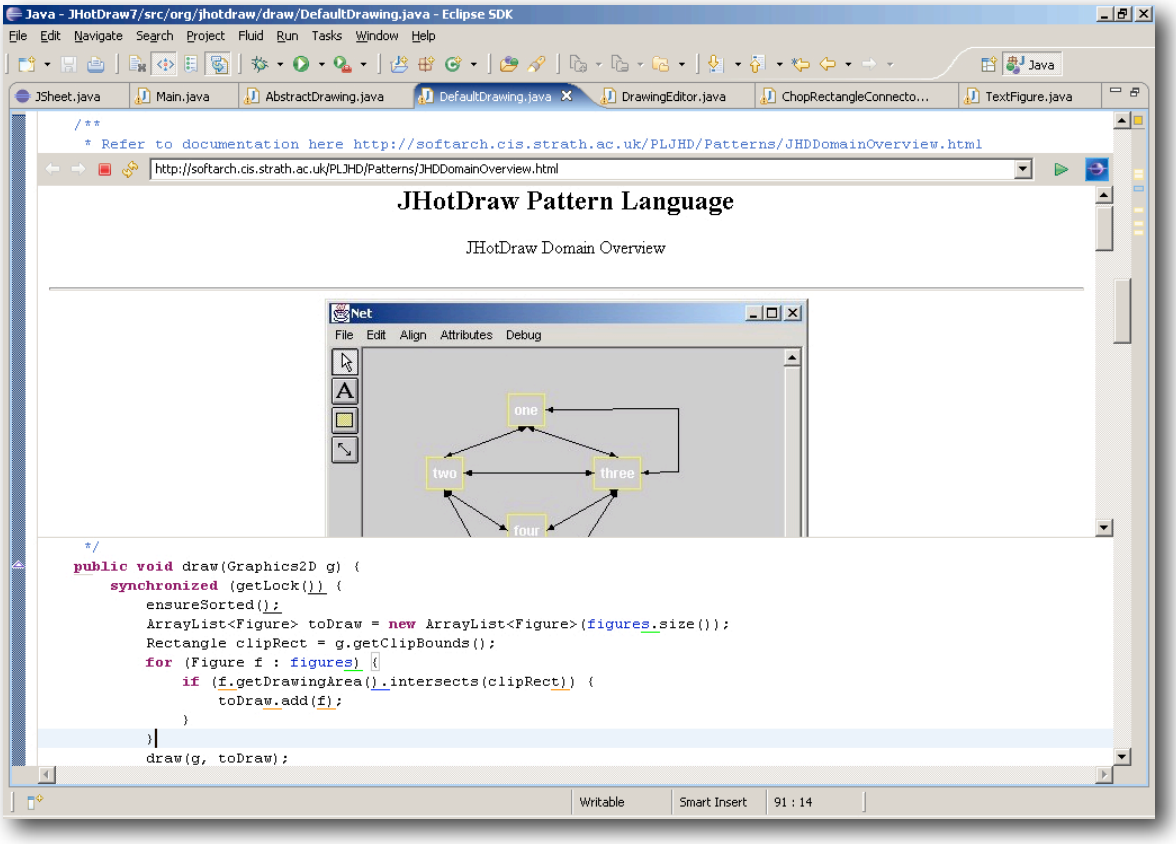


Figure 5.13: Activating the fluid annotation causes the associated web page to be introduced inline in a web browser instance.

5.3.2 Image resources

The fluid editor also supports the introduction of images directly into the source code context. When a source code document is opened, it is scanned for image references, namely literal strings which resolve to image files contained in the surrounding project or workspace. A fluid annotation is embedded at each image reference. When the annotation is selected, the corresponding image is introduced inline in a custom image viewer (See

Figure *5.14*). The image viewer supports the .gif and .jpg image formats and supports image zooming.

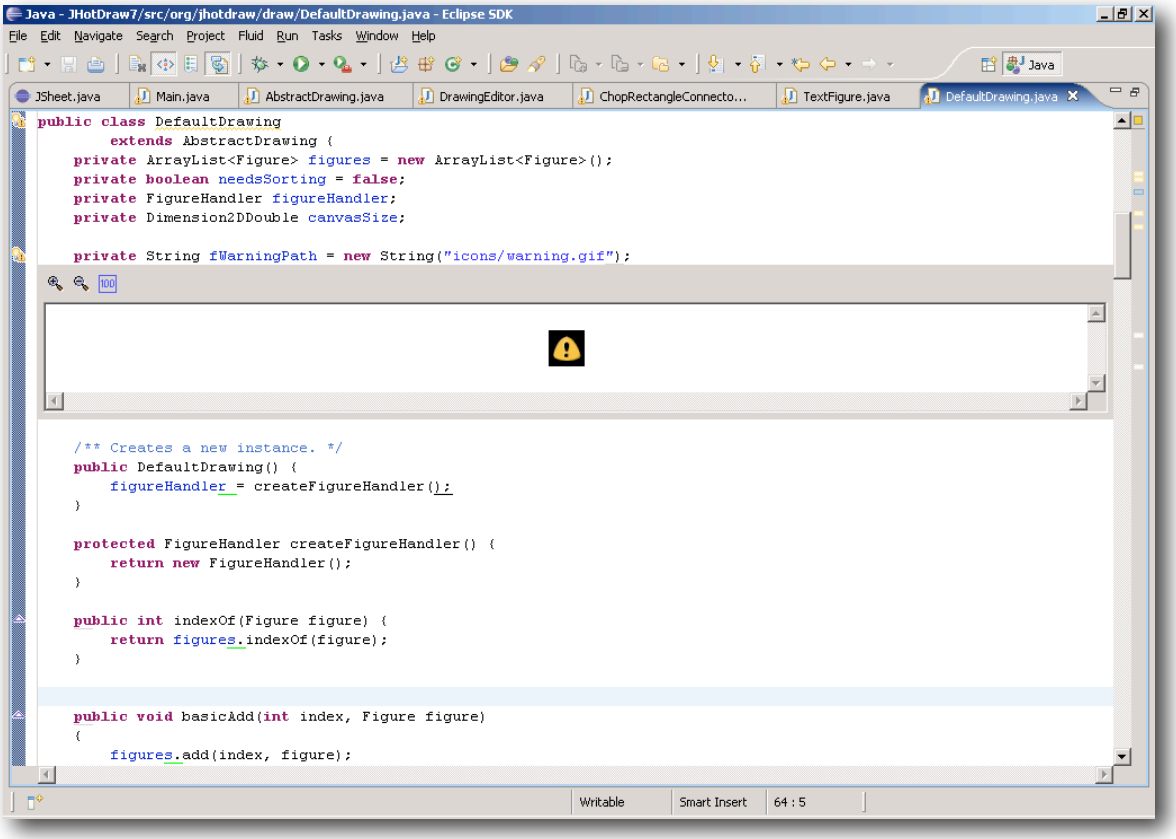


Figure 5.14: Inline introduction of an image resource. Images are introduced in a custom image viewer control which supports zooming.

5.4 Implementation

Under the covers, the fluid source code editor is essentially an extensible framework for annotating source code, and introducing external information into existing source code

documents. This section describes the implementation of the major components and interactions which make up the fluid source code editor.

5.4.1 Fluid annotations - generation, interaction and rendering

Upon opening a Java source code document the fluid editor initially creates an abstract syntax tree (AST) based on the document content via the AST libraries provided by the JDT. The AST is generated ‘with bindings’, meaning that the various references appearing in the AST may be resolved to their corresponding source code declarations.

Once generated the AST tree is visited or walked using a set of custom AST visitor classes designed to pick out references (nodes) representing potential inline exploration points. The selected references are checked to ascertain if the source code of the target declaration is available. References that do not have available source code are ignored as they are unsuitable for inline introduction. For instance, a reference to a method located in an external library without attached source code cannot be introduced as there is no physical source code declaration available.

A specific fluid annotation instance is created for each suitable AST reference. The annotation computes its position using the offset and length of the segment of source code corresponding to the reference, this information is subsequently mapped to the relevant pixel location in the fluid editor display during the rendering step. The fluid annotation is added to a fluid annotation model associated with the fluid editor instance. The fluid annotation model is responsible for maintaining the set of fluid annotations associated with a fluid editor instance and takes care of repositioning and deletion of annotations in response to edit events .

Rendering of fluid annotations onto the source code editor display is carried out by a fluid annotation rendering engine that is attached to the editor text widget. In response to paint events from the text widget the rendering engine iterates over all fluid annotations contained in the fluid annotation model and paints each annotation into the source code display. For efficiency the rendering engine culls annotations which lie outside of the current editor viewport. Each fluid annotation is responsible for painting itself onto the text widget. The abstract base class provides default painting behaviour which subclasses are free to extend or override as deemed necessary. Figure *5.15* presents a depiction of the fluid annotation life-cycle, from generation to rendering.

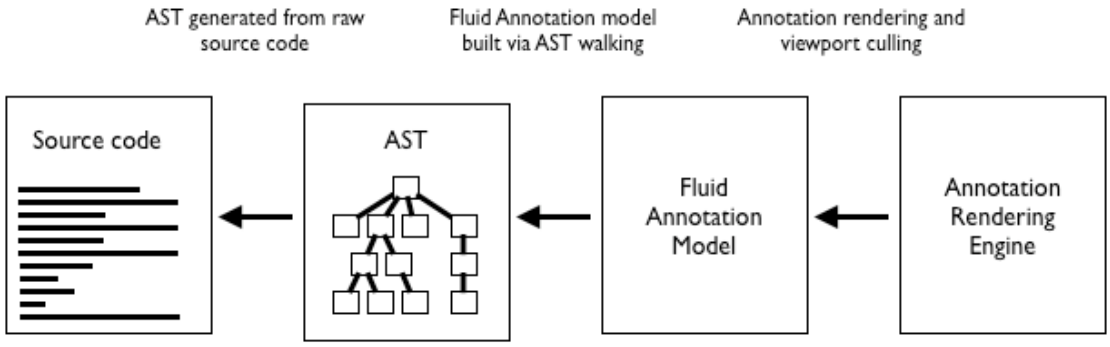


Figure 5.15: Fluid editor annotation generation, culling and rendering architecture.

The fluid editor manages interaction with fluid annotations by monitoring user interaction with the text widget onto which the annotations are painted. To simulate realistic interaction behaviour the fluid editor processes interaction events before they are passed to the text widget for default processing. In certain circumstances the fluid editor may prevent events from reaching the text widget - a behaviour referred to as ‘event hijacking’.

Interaction events are first relayed to the fluid editor which checks if the event affects any fluid annotations in the visible view port. If the event is deemed to affect a given fluid annotation, it is passed on to the fluid annotation itself for processing. The fluid annotation acts on the event and returns a boolean flag to the fluid editor indicating whether the event should be passed on to the text widget or discarded. An example of a potentially hijack-able event is the mouse down event. If the programmer clicks on a fluid annotation, the click event is not passed onto the text widget as this would make the interaction with the fluid annotation unrealistic. The event hijacking technique is vital to maintain a crisp and believable interactive document interface, making the fluid annotations seem like interactive elements rather than images painted on the canvas. Figure *5.16* illustrates the user interaction mechanism underpinning fluid annotations.

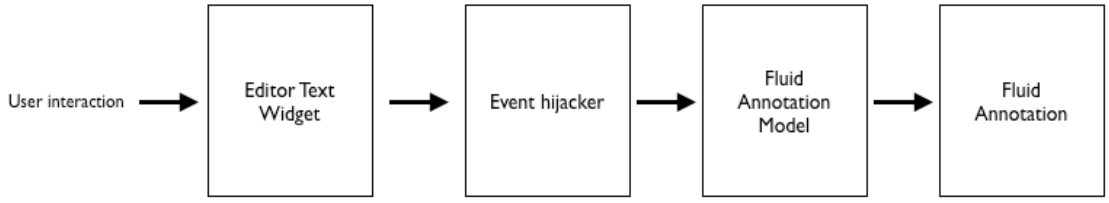


Figure 5.16: User interaction events may be hijacked via the fluid annotation model. A successful hijacking means that the event will not reach the editor text widget. This provides the user with a crisp interactive interface.

5.4.2 The dynamic document model

The fluid editor features the dynamic insertion and removal of source code declarations and other informational types to and from a given source code document. To implement this functionality an advanced dynamic document architecture was designed and implemented.

Each fluid editor maintains two documents in memory, a fluid document and a master document. The fluid document is presented to the programmer for editing and supports the dynamic inline introduction ‘fluid regions’. The master document is represents the structurally correct source code document that is used for compilation and is also written to disk when the editor buffer is saved.

The fluid and master documents are connected via a document information mapping (DIM). The DIM maps offsets, lines numbers and text regions from the fluid document to the master document and vice versa. Mapping from the fluid document to the master document essentially involves subtraction of any inline text regions above the mapping offset while mapping from the master document to the fluid document involves addition of any inline regions above the mapping offset.

Editing carried out on the fluid document is relayed back to the master document in real time. The fluid document edits are first mapped using the DIM and then applied to the master document.

Arbitrary regions of text may be added and removed from the fluid document programmatically using a custom insert/delete API. A fluid region contains an insertion offset and a formatted block of text which is to be inserted at the specified offset. The

insertion and deletion of a fluid region is essentially an edit of the fluid document (like a paste operation). However fluid region edits are explicitly prevented from synchronizing back to the master document (See Figure *5.17*).

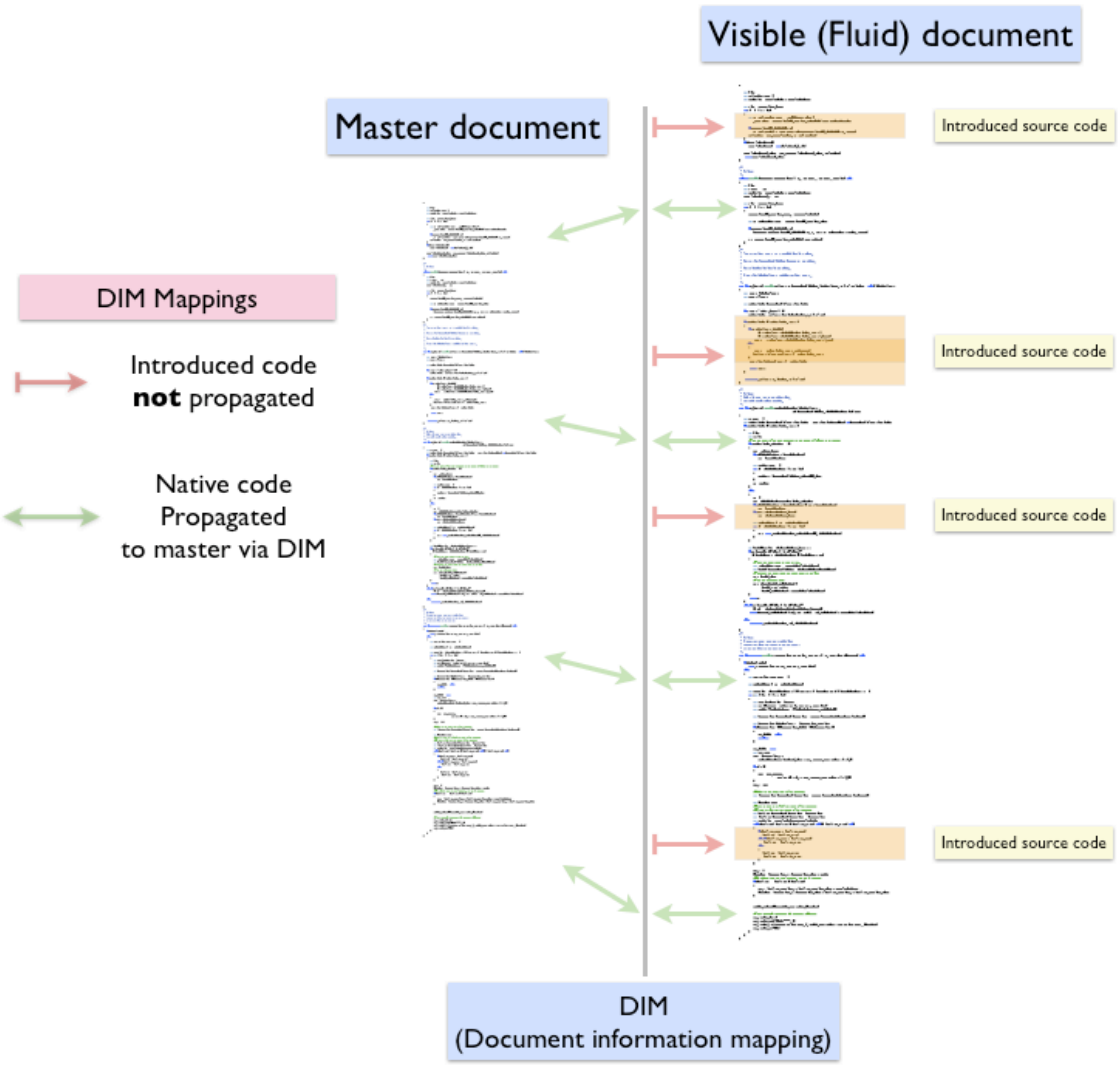


Figure 5.17: The fluid document architecture. The master document is maintained in a syntactically coherent state in memory for saving to disk and compilation. The visible or fluid document (presented to the user) may contain inline declarations (fluid regions) which are not propagated to the master. The Document information mapping (DIM) handles translation of offsets from the master to the visible document and vice versa. User edits and refactoring

operations carried out on the visible document are mapped in real time to the master document via the DIM.

Using the DIM as a synchronization bridge, the master document retains both its consistency with user edits and its structural integrity throughout fluid source code exploration activities. When the editor is saved the master document is written to disk, therefore the Java compiler sees the master document as opposed to the potentially non-compilable fluid document.

The inline introduction of non-text based content such a web browser instance or the image viewer is accomplished by inserting blank lines of text into the fluid document and overlaying a graphical control on the newly acquired space. The control is then synchronized with the editor window so that it is resized and moved in a consistent and believable manner.

5.4.3 Implementing an editing and reconciliation model

As previously mentioned editing and reconciliation is concerned with synchronizing the inline source code exploration model with changes to native source code.

On initialization each fluid editor instance adds a element changed listener to a central JDT model plug-in (JavaCore). The listener interface specifies an element changed method which is called when any element in the Java model (composed of all Java elements in the existing workspace) is changed, added or removed.

An element changed event received via the listener interface triggers a staggered refresh of the fluid source code model associated with each fluid editor instance. A refresh is a two step procedure consisting of first checking the validity and consistency of each inline declaration in the source code document, and then performing a refresh of the fluid annotation model.

The validity of an inline source code declaration is concerned with whether or not the declaration still exists in the workspace. Existence can be checked using the AST node associated with the declaration. If an inline declaration is deemed to be invalid, it is deleted from the source code document along with any contained inline declarations. The next step is to check consistency. The consistency of an inline declaration is determined by comparing the current contents of the declaration with a fresh copy of its native declaration. Inconsistent inline declarations are visually flagged to indicate the unreliable state to the user (See Figure *5.18*).

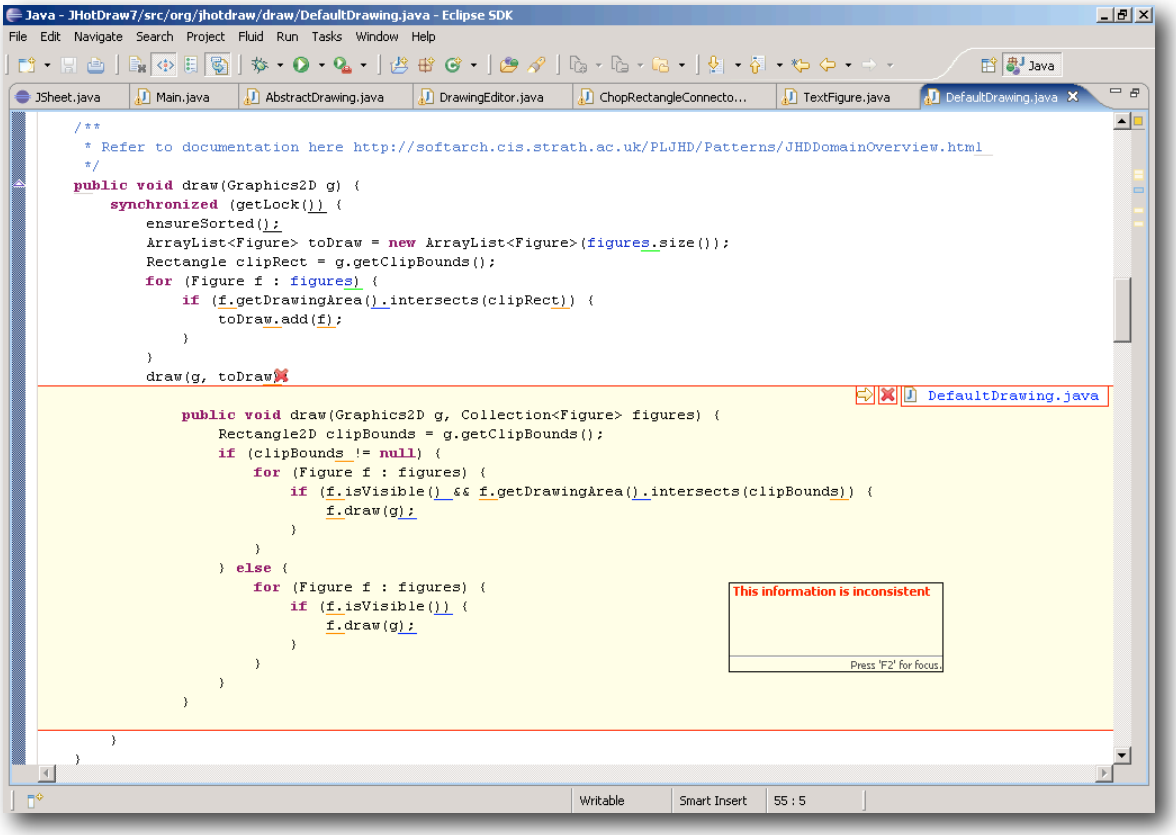


Figure 5.18: An inconsistent inline source code declaration, highlighted in red. Edits to the native declaration may cause inline copies to become inconsistent.

The refreshing of fluid annotations in response to element change events from the Java core model essentially involves a re-annotation of the source code document and any associated inline declarations.

5.5 Discussion

The fluid source code editor represents a fully functional inline source code exploration environment for Java source code. However the implementation of the system ended up

being a particularly arduous and time consuming task. To achieve the desired functionality, a significant amount of invasive extension and modification of the Eclipse IDE was required.

When the decision was made to develop an inline source code exploration interface, the choice of whether to build a simple throwaway prototype or attempt to extend an existing IDE offering with the desired functionality became a pivotal decision, affecting both the research and development effort. Although it involved a significant learning curve it was decided decided to build the system as an extension of the Eclipse IDE. This approach offered the best scope for realistic evaluation (performing a pilot study etc.) and offered the potential for a system usable and extensible by the wider development and research community. However as the fluid editor was under development, it became clear that both the Eclipse IDE and the JDT were not designed for such a significant extension efort, even when the rudimentary extension mechanisms provided by the platform were bypassed for greater control. To work around this limitation a number of core JDT and Eclipse plugins had to be invasively modified to add required features and make them more extensible. As a result, the fluid editor ships with a set of custom plugins which overwrite their core counterparts on installation (See Table *5.3*). Unfortunately, this means that the fluid editor only runs on a specific version of the Eclipse IDE (3.2). The system is also confined to the Windows platform, a side effect of the platform dependent nature of the standard widget toolkit (SWT) (Eclipse 2009g).

Core Eclipse and JDT plugins patched for Fluid source code editor compatibility
org.eclipse.jdt.ui
org.eclipse.jface.text
org.eclipse.swt
org.eclipse.swt.win32.win32.x86
org.eclipse.text

Table 5.3: The set of core Eclipse IDE and JDT plug-ins that required invasive modification and extension to support the fluid source code editor.

In addition to the extensibility problems, it was discovered that certain features of the Eclipse IDE are fundamentally incompatible with inline exploration. The most outstanding example is code folding (Eclipse 2009f). Code folding is standard feature in modern IDEs which allows the programmer to elide blocks of code in order to reduce clutter in the interface. For instance, the programmer can collapse comment blocks, methods bodies and even entire type declarations contained in a source code document in order to see more code of interest in a single display. Code folding essentially involves the removal of source code from the visible source code document, while fluid exploration involves the addition of source code to the visible document. Both technologies clash for control of the visible source code document and would need to know about one another to work in harmony. This integration work would have required a major redesign of the Java editor architecture which was deemed unfeasible within the context of the inline exploration project. As such, the fluid editor will ‘switch off’ code

folding when it is installed and will explicitly prevent the user from switching the feature back on.

Chapter 6

Experiment

The second major component of the research methodology, alongside the development of the fluid editor prototype, was the development and execution of a user experiment designed to determine if inline exploration was effective at reducing programmer disorientation during source code exploration activities. The goal was also to gather feedback related to the inline exploration approach, including the specific manifestation provided by the fluid editor.

The high level design of the experiment was that a number of participants, ideally with solid programming and IDE experience, would be recruited and asked to carry out a series of source code exploration tasks. Half of the tasks would be performed using the standard source code exploration interface provided by the Eclipse IDE (this would serve as a baseline level of disorientation), the other half would be completed using the inline interface provided by the fluid editor. During the tasks the level of disorientation experienced by the participants would be monitored and recorded using a variety of

qualitative and quantitate mechanisms. After the experiment, an analysis on the resulting data would be carried out, contrasting the level of disorientation experienced by participants on both of the interfaces.

The experiment was designed to be exploratory in nature. The aim was to observe the level of disorientation experienced on both interfaces and how users interacted with the inline source code exploration mechanism. However an informal preliminary hypothesis was derived. The expectation was that participants would experience considerably less disorientation using the inline interface (the Eclipse IDE with the fluid source code editor) versus the standard interface (the Eclipse IDE without the fluid source code editor). This chapter describes the design, rationale and execution of the experiment.

6.1 Soliciting feedback from the development community

Prior to the design and execution of the user experiment, an attempt was made to pre-evaluate the fluid source code editor via feedback from the developer community. The fluid editor prototype was released as an open source extension to the eclipse IDE, and made available a dedicated website describing the concept, features and potential advantages of the tool (Sourceforge.net 2009) (See Figure *6.1* or visit http://fluideditor.sourceforge.net). The project website also included a online survey allowing users to submit feedback and comments related to their experience using the tool.

While the fluid editor prototype enjoyed a significant number of downloads, it was found that little or no developers were interested in taking the time to fill in the online survey, or leave substantial comments concerning their experiences with the tool. Based on this lack of feedback it was decided that an exploratory user experiment was a more promising approach.

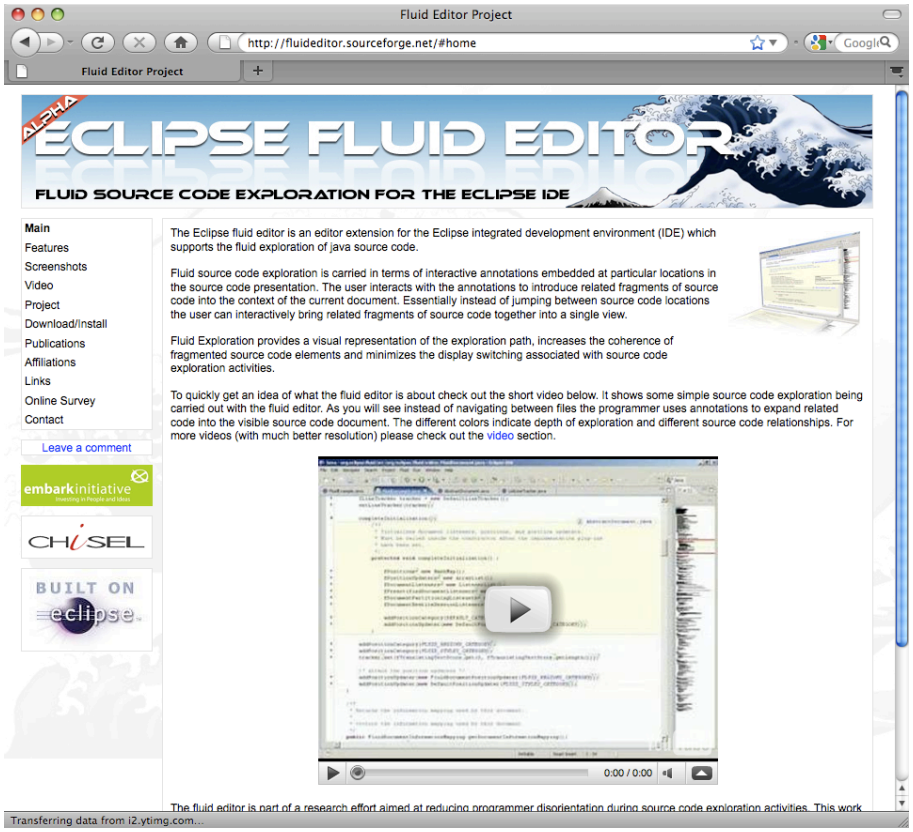


Figure 6.1: The fluid source code editor open source project website.

6.2 Measuring disorientation

To achieve the primary experimental goal it was vital to develop a credible and defensible mechanism by which to measure programmer disorientation during source code exploration tasks.

According to Yatim (2002) there are three mechanisms which are generally used to measure disorientation:

- Measuring degradation of user performance
- Gathering subjective feedback via questionnaires/interviews
- Examining the accuracy of the conceptual model assimilated by a user

Additional mechanisms deserving of consideration, but not described in Yatim 2002, are the observation of user behaviour for specific patterns or situations which are generally accepted to be indicative of disorientation (De Alwis & Murphy 2006) and measuring the level of visual momentum in the exploration interface (Watts-Perotti & Woods 1999).

- Observing the user to identify behaviour indicative of disorientation
- Measuring the degree of visual momentum in the interface

6.2.1 Measuring degradation of user performance

Edwards & Hardman (1999) point out that disorientation may be considered in terms of performance degradation, which is generally interpreted as the amount of time spent on the completion of a given task. The rationale is that a user who is experiencing disorientation will naturally attempt to reorient themselves, a process which requires additional time and effort. Therefore, it can be postulated that more time spent on a task the more disorientation experienced by the user.

Task completion time is a very attractive mechanism for measuring disorientation due to its simplicity, ease of collection and natural resonance in terms of user productivity. However, time should be interpreted carefully by the experimenter because the the link between time and disorientation is not very well understood. For instance, Maneti (1982) reported that disoriented users spent less time on tasks than those that were not disoriented. A severely disoriented user may simply quit their task out of frustration resulting in a significant reduction in completion time and an anomaly in the measurement technique.

6.2.2 Gathering subjective feedback via questionnaires/interviews

Perhaps the most basic approach to measuring disorientation is to ask users, via an interview or questionnaire, if they have experienced the phenomenon, and if so, to what extent (Maneti 1982). A popular mechanism is to ask the user to select from a scale of values indicating how lost or disoriented they felt during their task(s).

I often felt disoriented (lost) in the source code...

Disagree 0 1 2 3 4 5 6 7 8 9 **Agree**

However, gathering a subjective estimation of disorientation via an interview or questionnaire raises the issue of interpretation. Users may have a significantly different interpretation and understanding of the term ‘lost’ or ‘disoriented’ than the researcher running the experiment. Even the research literature does not unanimously agree on a definition. In some cases disorientation refers to a loss of spatial awareness (Conkin 1987) and in other cases losing ones train of thought due to the pursuit of digressions (Foss 1989a) and in other cases not being able to complete a goal (Watts-Perotti & Woods 1999).

The potential for misinterpretation raises questions about the relevancy and accuracy of information gathered from questionnaires and user interviews. However, the technique is still valid and useful to elicit general subjective feedback and thus should not be discounted.

6.2.3 Examining the accuracy of the conceptual model

The ‘conceptual model’ is the users mental representation and understanding of the information gleaned or assimilated during an exploration task.

It’s generally accepted that there is a link between the accuracy of a conceptual model and the level of disorientation experienced during its conception. For instance Elm and Woods (1985) define disorientation as when ‘the user does not have a clear

conception of relationships within the system'. Foss (1989a) maintains that an inability to devise an accurate conceptual model during browsing activities is a symptom of informational disorientation (the art museum phenomenon). It can be expected that the more accurate a user's conceptual model, the less disorientation experienced during its creation and refinement. The rationale is that disorientation interferes with the process of comprehension.

A simple mechanism for ascertaining accuracy of a conceptual model is to ask the user to sketch out the structure of the conceptual space associated with a particular task (Mahmoud 1993). The sketch can then be checked for errors or omissions against a known to be accurate depiction of the conceptual model.

However, accuracy of a conceptual model, in isolation, is not sufficient to measure disorientation. For instance, completion time is also a significant factor. A user spending a large amount of time exploring a body of information would be expected to produce an accurate conceptual model regardless of the disorientation experienced during the conception.

6.2.4 Observing the user to identify behavior indicative of disorientation

Perhaps the most promising overall approach to measuring disorientation is to observe how users carry out their tasks. In the research literature, there are numerous widely accepted interaction patterns and behaviours which are indicative of disorientation. For instance consider the following list of observable patterns:

- Backtracking to previously visited locations (Foss 1989a) (De Alwis & Murphy 2006)
- Pursuit of inefficient paths/loops (Foss 1989a)
- Restarting a task from a known location (Foss 1989a) (De Alwis & Murphy 2006)
- Closing all open files to clear context (Foss 1989a) (De Alwis & Murphy 2006)
- Thrashing between displays (Watts-Perotti & Woods 1999)
- Excessive interface adjustment (Watts-Perotti & Woods 1999)
- Failure to return from a digression (Foss 1989a)
- Failure to pursue a planned digression (Foss 1989a)

In addition to interaction patterns such as those presented in the list above, there are also specific gestures and comments which can be interpreted to indicate that the user is suffering from disorientation. Gestures might include drumming of fingertips, furrowed brows and puzzled stares (De Alwis & Murphy 2006).

Comments may also be considered and tend to be more specific than gestures. For instance, if one considers the characterization of programmer disorientation presented by De Alwis & Murphy (2006). A collection of interesting comments that are representative of disorientation may be identified:

- What am I looking at? - the programmer has lost their sense of location and direction in the code space.

- Why cant I find? - the programmer is unable to locate information of interest.
- What was I doing? - the programmer has lost their recent history.
- What was I going to do? - the programmer cannot remember their intent having arrived at a location.

Yatim (2002) provides a more comprehensive list of questions a disoriented user might be observed to ask themselves or allude to:

- Where am I?
- How did I get here?
- Where should I go next?
- I know where to go, but how to get there?
- What was I reading previously?
- What was I looking for?
- Why do I keep arriving at this page ?
- Have I been here before ?
- Is the information I am looking for available?
- This is not what I had expected
- I don’t think I found what I was looking for

Given the existence of interaction patterns, gestures and specific comments that may be linked to disorientation, it may be concluded that a promising strategy for measuring the

phenomenon would be to closely observe the user as they carry out their tasks. Then note down any situations which match the 'model' of disorientation. Users may be encouraged to comment via the use of a talk aloud protocol or perhaps using a more informal pair exploration type situation where the facilitator acts as a partner and sounding board for the participant, like a programming colleague might in a realistic setting.

It should be noted, however, that a synergistic approach is favourable to observing disorientation. An interaction pattern, gesture or comment in isolation may not indicate disorientation. A more interesting approach is to look for combinations or patterns which indicate specific types of disorientation. For instance, a programmer backtracking to previous locations with a furrowed brow and muttering about being unable to tell where they came from is a concise indication of navigation disorientation. On the other hand, a programmer backtracking might simply indicate the general needs of comprehending a complex piece of fragmented information

6.2.5 Measuring the degree of visual momentum in the interface

The degree of visual momentum in an interface may be used as a heuristic measure of disorientation (De Alwis & Murphy 2006; Watts-Perotti & Woods 1999). The greater the level of visual momentum in an interface, the less disorientation, it can be assumed, will be experienced by the user.

Woods (1984) describes an interface with low visual momentum -'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'. Based on this definition, it can

be concluded that a rudimentary mechanism to measure visual momentum is to record the number of display switches during an exploration task which involve a total replacement of visible content.

6.2.6 Discussion

Disorientation is a subtle, human-oriented and subjective phenomenon which means that it is fundamentally difficult to quantify in a defensible manner. Considering the variety of mechanisms available to measure disorientation, and the limitations associated with each, it was decided that the most appropriate approach was to use a combination. Essentially, the aim was to build up a model of disorientation based on as many factors as could be gathered during the experiment.

Quantitative metrics such as task completion time, display switches per task, amount of backtracking and interface adjustment were recorded. It was also observed how the participants interacted with the interfaces, their comments and gestures. A satisfaction questionnaire was developed which included a number of questions related to disorientation. Finally interviews were carried out in which the participants were allowed to describe their experiences using the interfaces in their own words.

6.3 Experiment design

Eight participants were recruited and asked to perform a series of eight small to medium scale exploration tasks over the source code of a moderately complex Java based drawing application.

A within subject design was used in the experiment. Each participant performed four tasks using the standard exploration interface (standard Eclipse IDE) and another four tasks using the inline interface (Eclipse IDE with the fluid editor installed). A significant advantage of the within subjects design approach is that it provides control of individual differences between participants. Essentially each participant acts as their own control group. Furthermore the design results in an effective doubling of the available data set in relation to a design organized around independent control groups. This is a considerable advantage due to the difficulties in finding willing participants.

However, carryover effects are a problem when applying a within subjects design. There are two basic types of carryover effects, practice effects and fatigue effects. When one within-subjects task negatively effects performance on a later task, this is referred to as a fatigue effect. It may be caused by factors such as tiredness or boredom. On the other hand, if one task is similar to another task, practice gained in the first task may lead to better performance in the second task, thus practice effects. Practice effects are a significant concern in this experiment because all tasks are based on a single underlying code base. To combat practice effects the order in which the tasks were performed and the order in which the participants used the interfaces were systematically varied to counter-

balance any potential skew. While this would not eliminate the effects it would distribute them evenly over the resulting data.

The tasks were recorded using a screen capture device and a video camera. A 'think aloud' approach was used to elicit comments from participants as they carried out the source code exploration tasks.

6.3.1 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 (See Table *6.1*).

Participant	Profession
P0	Faculty at the University of Limerick
P1	Part time M.Sc. Student/Professional programmer
P2	PhD graduate/Professional programmer
P3	Faculty at the University of Limerick
P4	PhD student
P5	PhD student
P6	PhD student
P7	PhD student

Table 6.1: Participants who took part in the exploratory user experiment and their corresponding profession.

All participants involved in the study were required to have strong programming experience, however, Java language experience and experience using the Eclipse IDE varied significantly over the participant set. 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 (See Table *6.2*).

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	Experienced

Table 6.2: Participants and their experience with Java and the Eclipse IDE. The data was transcribed from a profile questionnaire filled out by each participant at the beginning of their experiment session.

Ideally, the study should have involved only those participants who were fully comfortable with both Java and the Eclipse IDE, however due to limited time and availability of willing participants a compromise had to be made. The primary issue with using in-experienced participants is an increased tendency towards disorientation, not based on the structure of the exploration interface itself but rather the process of adjusting to the new environment, source code and programming language.

6.3.2 Tasks

A significant design element associated with the experiment was the selection of tasks. Initially it was decided to pursue as realistic an approach as possible and have participants carry out live maintenance on a system. The participant would be presented with a description of a particular issue of enhancement and would need to identify and explore the relevant code and make the necessary changes.

However, during the initial pilot study, involving two participants, which was carried out before the main experiment, it became clear that asking participants to perform live maintenance was problematic. Due to inexperience, the participants took a significant amount of time to perform the relatively small maintenance tasks, and seemed to undergo a significant amount of stress and confusion in the process. Unfortunately the study did not have access to professional programmers and relied on graduate students who exhibited varying degrees of programming expertise and tolerance for complexity.

Based on the experience of the pilot study, it was decided to use basic source code exploration tasks, not involving live maintenance on a system. The potential for ruining the experiment was too great to use realistic maintenance tasks.

The revised source code exploration only tasks were designed to address both navigation of source code and development of a conceptual model, activities which are inextricably linked to the disorientation phenomenon. Each task was structured as a series of questions which related to particular area of the system. The participant was asked to

read each question then explore and comprehend the associated source code and verbally provide answers to the experiment facilitator.

Tasks were categorized into four different types, each of which was targeted towards a particular source code exploration scenario (See Table *6.3*). During each experiment session the participant carried out one task of each type on both exploration interfaces. The use of task types was designed to coincide with the use of the within subject approach. Because the study would be comparing the results of a single participant using both interfaces it was felt that it would provide greater accuracy to compare similar task types.

Task Type	Inline Interface	Standard Interface
Local Neighbourhood	1	1
Control flow	1	1
Polymorphic	1	1
Inheritance	1	1
Total	**4**	**4**

Table 6.3: Task types carried out on each interface.

6.3.2.1 Local neighbourhood task

The local neighbourhood tasks involved the exploration 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 within three navigation steps. The neighbourhood essentially represented a radius of navigable code.

The local neighbourhood tasks emphasized source code navigation and development of a conceptual model. 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 digression. 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.

6.3.2.2 Control flow task

The control flow tasks focused on the navigation and comprehension of a complex chain of scattered control flow encompassing source code from a number of locations and documents across the code space. The average number of locations involved in a control flow task was eight. The participant was guided to a particular location in the code and asked to examine the structure and logic of a program operation driven by a number of informational goals.

The control flow tasks emphasized the exploration comprehension of fragmented source code and navigation through a complex control flow chain with potential for digressions. 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 space. The participant was asked to describe certain

aspects of the operation as well as to provide a high level description of the logic and overall functionality.

6.3.2.3 Polymorphic task

The polymorphic tasks focused on the exploration and comprehension of a given abstract program 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. The tasks involved comparison of declarations and analysis of the type structure. The participant was also required to explore further into code space beyond the declarations.

The primary aim of the polymorphic tasks were to determine how the interfaces performed when abstract operations were encountered and the participant would need to make use of search functionality. It was of particular interest to see how participants would react to the inline search feature and how it would compare to the comparative approach provided by the standard interface.

6.3.2.4 Inheritance task

The inheritance tasks 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 participants were 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.

The inheritance tasks emphasized the navigation of inheritance relationships and comprehension of source code located at various levels of the type hierarchy.

6.3.3 Data

To measure the level of disorientation experienced during the completion of the exploration tasks, and also gather feedback regarding the usage of the inline interface, the study involved the gathering of both quantitative and qualitative data during the experiment.

The quantitative data gathered during the experiment included:

- Task completion time
- The number of display switches per task (exhibiting a total replacement of content)
- The amount of backtracking carried out per task
- Interface adjustment (Scrolling)
- Number of inline introductions per task

Quantitative data was gathered using a monitor installed on the version of Eclipse used during the study. The monitor was built as an Eclipse plug-in and included a simple task

view containing the list of tasks associated with the experiment session. Each task entry had a start and stop button. When ready to begin a particular task the participant selected the start button which initiated the monitor.

The monitor logged the duration of the task along with the number of display switches, use of the history stack, introductions and various events associated with interface adjustment. This information was stored in a file labelled with the selected task name. When the task was complete the participant was asked to press the stop button on the active task which stopped the monitor from logging further events and closed the log file associated with the task.

The qualitative data gathered during each session included:

- Observed behaviour, comments and gestures
- Satisfaction questionnaires associated with each interface
- Exit interview

Each session was recorded via a screen capture tool and a video camera. The screen capture tool recorded the entire screen as the participant carried out the exploration task. The video camera was placed behind the programmer and slightly to the right. This angle captured any comments made by the participant and the facilitator as well as the participants body language and the screen.

The participant was not asked to talk aloud, a process that could be considered as being overly artificial, but was instead encouraged to talk informally about the task using the facilitator as a sounding board for any thoughts or concerns. Essentially the participant was encouraged to discuss the task as they would with a quiet colleague. The facilitator sat beside the participant and to the right, careful not to block the view of the camera. The facilitator monitored the participant and noted down any interesting behaviour, comments or gestures which might indicate disorientation.

After each set of exploration tasks, the participant was asked to complete a satisfaction questionnaire. The questionnaire contained a variety of questions based on their satisfaction with the interface they had just used. Questions were based on the work of Chin *et al.* (1988) and Jakobson & Hornbaek (2006). The questionnaire also included a number of questions specifically focused on ascertaining the level of disorientation experienced. At the end of the session the participant took part in an exit interview during which they were allowed to reflect on the experience in their own words.

The details of the questionnaires and the exit interviews are discussed in the results section so as to avoid needless repetition.

6.3.4 Environment

The experiment was carried out in a laboratory setting using a laptop computer (Thinkpad X31) attached to a single 17 inch LCD monitor. Screen resolution was set to 1024 x 768. Participants used an external USB mouse and keyboard for interaction with the system.

6.3.4.1 Software

Eclipse version 3.3 was used as the overall experiment platform. The standard exploration interface was the basic Eclipse 3.3. The fluid source code editor version 1.1.0 was used as the interface for the inline exploration tasks (See Table *6.4*).

Interface	Software configuration
Standard	Eclipse 3.3
Inline	Fluid Editor 1.1.0 installed on Eclipse 3..3

Table 6.4: The software configuration used during the experiment.

The main Eclipse window was presented in full screen mode occupying all of the available screen space. By default the package explorer, the console view and the outline view were open and visible. 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. The fluid editor was presented with default settings (standard color model and shading to differentiate between levels in the inline exploration tree).

6.3.4.2 JHotDraw

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 which includes a basic drawing editor as a sample application. The JHotDraw source code is relatively small but moderately complex which offered a good balance in terms of task complexity and expected completion time.

6.4 Procedure

The following section gives an account of how the experiment was carried out from a procedural point of view. This particular procedure was repeated eight times over the course of the experiment, once for each participant. The overall experiment was carried out over a four week period.

Upon entering the laboratory, the participant was first welcomed and thanked for taking part in the study. The facilitator then delivered a ten minute high level description concerning the organization of the experiment and what was expected of the participant. The intent of the experiment was purposefully omitted including any reference to disorientation. It was felt that such information might introduce preconceptions and alter the participants natural behaviour. After the initial introduction, the participant was asked to fill out a profile describing their Eclipse and overall programming experience.

Once the preliminaries were out of the way, the participant was introduced to the first interface that they would be using during the experiment. This was either the

standard interface or the inline interface depending on the configuration of the particular session (interface order was systematically varied over the course of the experiment to counter skew due to practice effects). The primary features of the interface were demonstrated, such as how to navigate within the source code, how to open files and how to use the primary IDE views. The participant was encouraged to spend at least ten minutes using the interface in order to get comfortable and clear up any potential usability problems.

Once the participant had confirmed that they were indeed comfortable with the interface and had no further questions, the first task sheet was produced. The task sheet included a description of the required task. The participant was asked to read the task description and indicate to the facilitator when they were ready to begin exploration.

On commencement of the exploration task, the facilitator asked the participant to select the task in the task view and click the start button. The video camera was also started as synchronously as possible using a remote control. The participant completed the exploration task using the facilitator as a sounding board. The facilitator encouraged comment when necessary by asking the odd non-confrontational question and making innocuous comments concerning the exploration task. The facilitator sat to the right and a little behind the participant and monitored the exploration and recorded any interesting interaction patterns, comments or gestures throughout the course of the task.

Upon completion of the exploration task, the facilitator asked the participant to stop the active task using the task view. The video recording equipment was also paused.

The participant was allowed a short break and asked to indicate to the facilitator when they felt ready to begin the next task. All four tasks were carried out in this exact manner.

When the four tasks on the initial interface were completed the participant was asked to fill out the satisfaction questionnaire associated with the interface. The facilitator left the lab for ten minutes allowing the participant to fill in the questionnaire in a private and pressure free setting.

Upon completion of the questionnaire, the participant was introduced to the second exploration interface and carried out the same sequence of steps associated with the first interface, including the completion of a corresponding questionnaire.

Before the end of the session the participant took part in the exit interview. The exit interview allowed the participant to describe their experience in their own words.

After the interview the participant was thanked again and shown out of the laboratory. The overall experiment duration was expected to be 2 - 2.5 hours.

Chapter 7

Results, Findings & Validity

The user experiment resulted in a significant amount of data including monitor log files, video and screen capture recordings, questionnaire results, interview transcripts and observational notes. This body of raw data was analysed and the results and findings are presented in this chapter. This chapter also discusses the validity of the experiment.

7.1 Task completion times

Task completion times were calculated from the monitor log files generated during the experiment. Overall participants completed the exploration tasks 14 % faster on the inline 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) per 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 7.1: Task completion times in tabular format (STD = standard deviation). All values are in seconds.

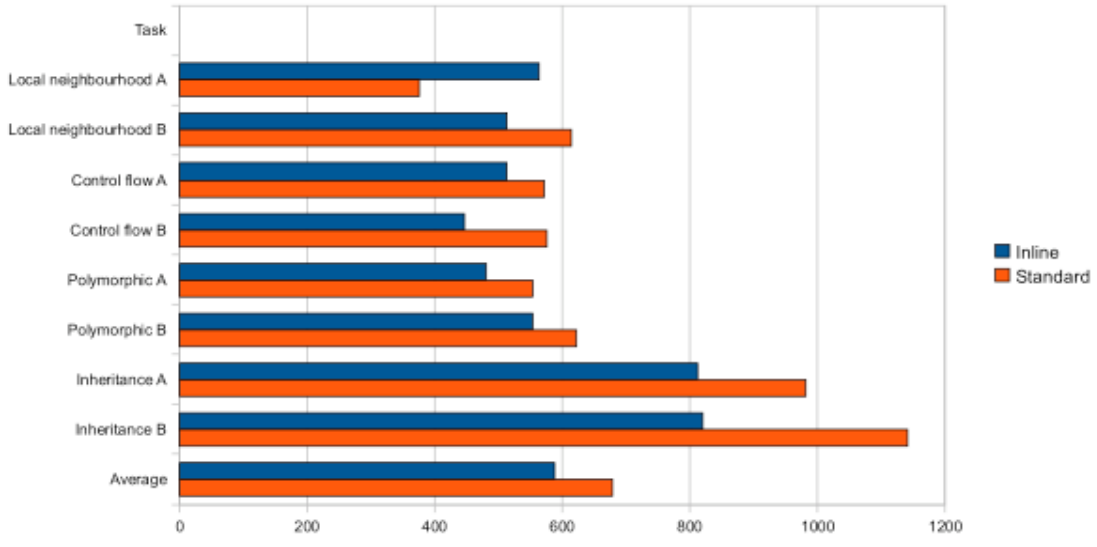


Figure 7.1: Task completion times in bar chart format.

Participants completed local neighbourhood task 8% faster using the standard interface versus the inline interface. This is the only task in which the standard interface yielded a faster completion time. The first local neighbourhood task (A) was performed 33% faster using the standard interface however the second neighbourhood task (B) was performed 17% faster using the inline interface.

Participants performed the control flow task 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).

With regard to the polymorphic tasks, participants performed 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, participants completed 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 and 14% faster using the inline interface. The results indicate a general trend of improved completion time on the inline interface.

7.2 Display switches

The number of display switches, in which a total replacement of visible content occurred, was used as a heuristic measure of visual momentum in the exploration interface. The greater the number of display switches during a task the lower the level of visual momentum and vice versa.

Overall the average number of display switches per task was 95% lower with the inline interface versus the standard interface.

	Inline		Standard	
Task	**Mean**	**STD**	**Mean**	**STD**
Local neighbourhood A	3.8	5.2	17.8	6.2
Local neighbourhood B	0	0	15.3	9.5
Control flow A	2	1.6	13.3	5.9
Control flow B	0.5	1	22.3	8.7
Polymorphic A	0	0	15	6.7
Polymorphic B	0	0	12.3	1.9
Inheritance A	2.3	2.6	37.8	22.2
Inheritance B	0.3	0.5	43.8	6.3
Average	**1.1**	**1.4**	**22.2**	**8.4**

Table 7.2: Display switches per task in tabular format (STD = standard deviation).

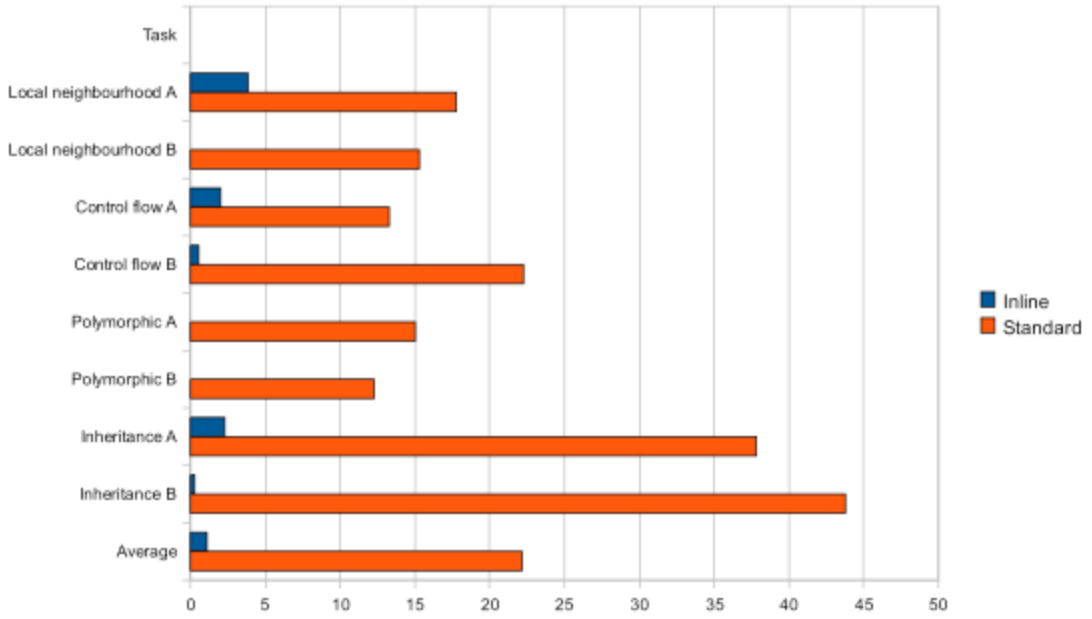


Figure 7.2: Display switches per task in bar chart format.

Over the local neighbourhood task set participants experienced an 89% reduction in display switching using the inline interface. On the control flow tasks participants experienced a 93% reduction with the inline interface. On the polymorphic tasks participants did not, on average, experience any display switching when using the inline interface. Completing the polymorphic tasks on the standard interface an average of 14 display switches occurred. Over the inheritance tasks participants experienced a 97% reduction in display switching.

In addition to the number of display switches per task, the number of inline introductions was also recorded for each task carried out on the inline interface. Using the number of introductions it is possible to get an idea of the number of navigational actions carried out by the programmer over the task sets. 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 per task. Based on this data it was discovered that participants perform 31% less navigation actions using the inline interface.

	Inline		Standard
Task	**Introductions**	**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**
Navigation actions	**15.4**		**22.2**

Table 7.3: Navigation actions carried out per task in tabular format (STD = standard deviation).

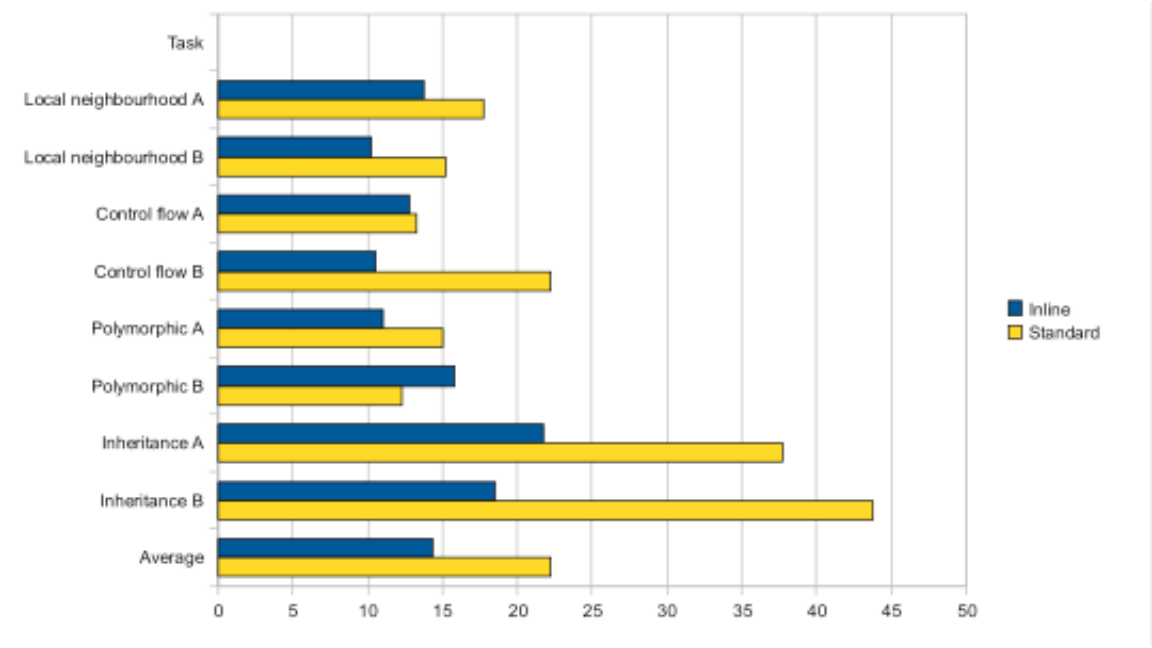


Figure 7.3: Navigation actions carried out per task per task in bar chart format.

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 and indicates the existence of an outlier. 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.

7.3 Backtracking

On the inline browsing interface the use of the back and forward navigation actions was negligible. Over the entire data set of 32 tasks the forward action was invoked three times and the back action was invoked 15 times. This represents a 0.6 backward actions per task and 0.3 forward actions carried out per task. On the standard interface the back action was invoked, on average, 6 times per task and the forward action was invoked 1.6 times per task.

	Inline		Standard	
Task	**Back**	**Forward**	**Back**	**Forward**
Local neighbourhood A	1.75	0	4.3	0
Local neighbourhood B	0	0	4.3	0.8
Control flow A	2	0.75	4.8	3.5
Control flow B	0	0	6	3.3
Polymorphic A	0	0	2.8	0
Polymorphic B	0	0	1.8	0
Inheritance A	0.75	0.75	15	3.5
Inheritance B	0.5	0.5	8.8	1.8
Average	**0.6**	**0.3**	**6**	**1.6**

Table 7.4: Mean backtracking over the task set.

The results indicate a 90% reduction in backward navigation and an 81% reduction in forward navigation when using the inline interface.

7.4 Scrolling

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

	Inline		Standard	
Task	**up**	**down**	**up**	**down**
Local neighbourhood A	49	71	17.3	76.5
Local neighbourhood B	17.3	21.8	11	54
Control flow A	59.3	126	15.8	50.3
Control flow B	54.8	82.5	45	36
Polymorphic A	10	26.3	13.5	60
Polymorphic B	9	25.5	19	24
Inheritance A	38	136.5	43.5	136.5
Inheritance B	87.8	215	9	24
Average	**40.7**	**88.1**	**21.8**	**57.7**

Table 7.5: Vertical scrolling carried out per task.

7.5 Satisfaction

All questions in the satisfaction questionnaire were answered on a scale of zero to nine forcing the participant to lean to one side of the scale. All questions in all of the satisfaction questionnaires were completed. The results of the satisfaction questionnaires are presented in Tables *7.6* & *7.7*.

	Inline		Standard	
Question?	**Mean**	**STD**	**Mean**	**STD**
1. How did you find the inline browsing interface in general?				
Very poor - 0 1 2 3 4 5 6 7 8 9 - Very good	7.7	1	5.4	1.4
2.- 6. How was the interface to use?				
Terrible - 0 1 2 3 4 5 6 7 8 9 - Wonderful	7.3	0.8	5.1	1.6
Hard - 0 1 2 3 4 5 6 7 8 9 - Easy	7.6	1	4.4	1.1
Frustrating - 0 1 2 3 4 5 6 7 8 9 - Pleasant	7	1	4.4	1.1
Boring - 0 1 2 3 4 5 6 7 8 9 - Fun	7.4	1.3	5.1	1.5
Confusing - 0 1 2 3 4 5 6 7 8 9 - 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 - 0 1 2 3 4 5 6 7 8 9 - I agree	7	1.4	4.6	1.7
8. I often lost my orientation (got lost) in the source code.				
I disagree - 0 1 2 3 4 5 6 7 8 9 - I agree	4.7	2.7	5.7	1.7
9. I often felt confused when exploring the source code.				
I disagree - 0 1 2 3 4 5 6 7 8 9 - I agree	4.9	2	5.4	1.1
10. There was sometimes too much information on the screen at once.				
I disagree - 0 1 2 3 4 5 6 7 8 9 - I agree	5.6	2.5	4.1	2.4

Table 7.6: Satisfaction questionnaire results, questions 1-10.

Question?	Inline		Standard	
	Mean	STD	Mean	STD
11. It was easy to determine the relationships between expanded pieces of code.				
I disagree - 0 1 2 3 4 5 6 7 8 9 - I agree	5.7	2.4		
12. The visual cues were distracting...				
I disagree - 0 1 2 3 4 5 6 7 8 9 - I agree	2.3	2.7		
13. The colouring coding of the visual cues was helpful..				
I disagree - 0 1 2 3 4 5 6 7 8 9 - I agree	5.1	2.3		
14. How did you perceive the tasks?				
Very poor - 0 1 2 3 4 5 6 7 8 9 - Very good	5.9	1.3	6.1	1.1
15. How would you rate your answers to the tasks?				
Very poor - 0 1 2 3 4 5 6 7 8 9 - Very good	4.7	2.4	4.9	1.6
16. - 18. Was the source code...				
Hard to understand - 0 1 2 3 4 5 6 7 8 9 - Easy to understand	4.6	1.7	4.1	2.1
Hard to overview - 0 1 2 3 4 5 6 7 8 9 - Easy to overview	4.4	2.1	4	2.6
Hard to navigate - 0 1 2 3 4 5 6 7 8 9 - Easy to navigate	6	2.4	4.1	1.9
19. Was information in the source code...				
Hard to locate - 0 1 2 3 4 5 6 7 8 9 - Easy to locate	6.3	2	4.4	2.3

Table 7.7: Satisfaction questionnaire results, questions 11-19. The grey cells indicate questions that were specific to the inline interface.

The results of the satisfaction questionnaire were analysed for trends in the data, the results are presented here. Overall participants preferred the inline browsing interface over the standard interface. The inline interface scored better on the scale of terrible to wonderful. Participants also found the inline interface easier to use, more pleasant, more fun and less confusing.

Participants agreed that they had a better idea of where they were in the code using the inline interface. The data also suggested that participants felt less disoriented using the inline interface. Participants felt less confused using the inline interface. Interestingly the data suggested that participants found that there was too much information presented using the inline interface.

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

The data suggested that participants found the it slightly easier to comprehend code, overview code and navigate code using the inline interface. Participants also agreed that it was easier to locate information in the source code using the inline interface.

7.6 Exit questionnaire

The exit questionnaire contained a series of five open ended questions encouraging the participant to reflect on their experiences during the study. The facilitator transcribed participant responses. A summary of each question coupled with interesting and relevant responses is presented in this section.

Did you prefer the inline interface or the standard interface? Please elaborate.

All participants indicated that they preferred the inline browsing interface over the standard browsing interface. Participants were also asked to elaborate on their decision.

Participant P0 indicated that, initially, the concept of inline exploration was difficult to grasp but once comfortable the technique was described as ‘intuitive’. P0 also mentioned that he suffered from colour blindness and thus some of the annotation and background colours looked alike.

P1 described the fact that ‘the control flow path was easy to overview’ and that the inline interface was ‘simple and convenient for following control flow’. Adding to this P1 said that when using the standard interface he tended to ‘miss things and forget information’. P1 also mentioned that the inline search results view was ‘good’ and that the inline browsing interface was ‘excellent for exploring other people’s code’. Finally P1 mentioned that the inline interface required ‘no adjustment’ and was something he would ‘use all the time’.

P2 indicated that the inline interface ‘immediately made sense’ was ‘more fun’ and that it ‘fixed many of the problems with the IDE’. P2 also communicated the desire for the ability to collapse certain levels in the inline browsing tree while keeping the sub tree open.

P3 stated that the inline interface made it ‘easier to keep track of locations’ and that he liked how the programmer ‘stayed in the single view’.

P4 mentioned that it was ‘easier to find the code that you are looking for’ and that the code displayed using the inline interface was ‘clearer than standard mode’. P4 also said that the inline interface involved ‘less jumping around code’ and was ‘more intuitive’.

P5 described that the ‘consistent’ interface of the inline interface was useful. He indicated that ‘with standard Eclipse there are many ways of finding and navigation’ but with the inline interface there was a ‘consistent approach’. P5 also mentioned that one could ‘look at other code without loosing your position’ and that the code was ‘easier to navigate’ using the inline interface. P5 went on to say that the inline interface supported ‘comparison of different places’ and that it was “easy to follow multiple steps through code”.

P6 mentioned that it was easier to ‘back out of wrong turns’ using the inline interface and that he could ‘see the exploration path’. P6 added that it was ‘hard to follow sequences of code using standard Eclipse’.

P7 did not take part in the exit interview due to the experiment session overrunning the allocated time.

What advantages or disadvantages did you perceive when using the inline interface vs. the standard exploration interface?

The second question asked participants to describe the advantages and disadvantages of the inline interface versus the standard interface. In hindsight the question was poorly worded as some participants, in their answer, described advantages and disadvantages of the inline interface as opposed to comparing both interfaces.

When asked to describe the advantages and disadvantages of the inline interface versus the standard interface P0 mentioned that one does not need to ‘leave the context’

and that the inline interface provides a ‘nice summary of explored elements’. When asked about disadvantages P0 said that the ‘UI was jumpy’.

In terms of advantages P1 indicated that the visual cues ‘don’t get in your way’ and that the colour coding of inline source code was ‘useful’. Describing disadvantages P1 indicated that the relationships between inline declarations were sometimes ‘difficult to discover’ and that it was not possible to close inline declarations using the keyboard.

P2 provided a detailed list of advantages but was unable to provide any disadvantages. The advantages were:

- ‘The inline interface made sense of abstract classes/interfaces’
- ‘Inline exploration shows you the code you are interested in without any hassle’
- ‘Good for taking a quick look at code without leaving’

P2 also mentioned that the use of ‘multiple editors’ in Eclipse ‘is a cheap fluid editor’. According to P3 the advantage of the inline interface is that it ‘facilitates comparison of sequential and hierarchical code’. In terms of disadvantages P3 mentioned that the inline editor was not good for ‘comparing code that was not linked’ and that the colours were ‘arbitrary’.

P4 declined answering this particular question stating that he needed ‘more experience with Eclipse’ before he could provide a proper answer. P5 said that the visual

cue colouring should be ‘ignored’ but that the shading colour model was ‘good to convey depth’.

Finally P6 described the advantage of the inline interface as simply ‘seeing everything at once versus paging through’. ‘Visual cues interfering with each other’ was described as a disadvantage as was the visual cue colour model and the fact that visual cues would ‘sometimes blink when reading the code’ serving as an annoyance to the programmer.

Can you identify any aspects of the inline interface that stood out as particularly confusing or frustrating?

When asked to identify aspects of the inline interface which stood out as particularly confusing or frustrating P0 mentioned that ‘visual cues are sometimes hard to click on’ but otherwise everything was ‘straight forward’.

P2 described that it was difficult to ‘differentiate between adjacent visual cues’ and that he ‘didn’t pay attention to the colors’. P2 also indicated that deep inline exploration trees were not ‘too difficult’ to deal with.

P3 again mentioned that it was ‘hard to find the right cue and that when a lot of information was introduced inline reading the code was ‘difficult’. All other participants indicated that they could not think of anything which stood out as particularly confusing or frustrating.

In hindsight the content of question three overlapped with the previous question and thus it was not surprising that some participants were unable to describe any

confusing or frustrating aspects of the inline interface. These aspects had already been described in question two and thus, on some levels, question three was superfluous.

Are there any features that you would like to see in future versions of the fluid editor?

When asked to identify desired features of the fluid editor or an inline interface in general P0 said that he would like to be able to edit the code contained in inline declarations. When asked why, P0 replied that it was an 'obvious feature but maybe not needed?'.

P1 requested the 'ability to introduce the call hierarchy and view search results inline'. P1 also mentioned the ability to 'hide levels of the hierarchy'. P2 wanted to see inline introductions surrounded in a border and indented out from the horizontal axis of the editor. He also requested a differentiation between 'normal' code and 'interfaces'. P3 described the need for a visual map of the path associated with the exploration task. P5 said he would like to 'see the editing of inline introductions just out of curiosity'. Finally P6 wondered if there would be any advantage to support for saving inline browsing paths.

Would you see yourself using the fluid editor or a similar system when programming?

All participants indicated that they could see themselves using an inline browsing interface if it was provided as part their IDE. P0, who does not use an IDE, indicated that he would use an inline interface if he was using an IDE.

7.7 Findings

In addition to the results presented in the previous sections, numerous pages of observations were captured concerning participant behaviour, interactions, comments and gestures over the course of the experiment. Furthermore, after the experiment was complete in a procedural sense, the various notes, screen captures and video recordings were synchronized and analysed in detail to clarify findings and carry out a more focused observation beyond that possible during the live sessions.

Rather than providing an exhaustive listing of observations and incidents as part of the results section, this information is instead presented in a condensed form as part of the findings. Findings are organized into two broad sections. First a discussion of the findings related to disorientation from observing participants working on the standard interface. This insight proved to be extremely useful both in terms of gaining a greater understanding of programmer disorientation in the IDE and also adequately evaluating the inline approach. Secondly a discussion of the effectiveness of the inline interface at alleviating the observed disorientation. For this discussion the results presented in the previous sections in addition to the various observations gathered over the course of the experiment sessions are drawn upon.

7.7.1 Disorientation observed using the standard interface

During the experiment a deep insight into disorientation as it occurs during source code exploration activities using the standard Eclipse IDE was gained. Based on the various

observations, a set of core factors and scenarios which typified the phenomenon were formulated. It should be noted that this set of factors is not considered comprehensive as it is based on observations from the study itself, which was limited in both size and scope.

7.7.1.1 Navigation context

Over the course of the experiment it was observed on a number of occasions, that participants would suffer from disorientation due to a lack of navigational context. This coincides with the findings of Murphy *et al*. (2006) and a general consensus in the overall research literature. At a basic level, a lack of navigational context means that there is no visible representation of the navigation path which the programmer pursued to arrive at the currently visible source code location, an item of context which is important in terms of reminding the programmer of their ongoing intent and task focus (Storey *et al.* 1999; De Alwis & Murphy 2006).

Participants were observed, often during complex navigation activities, having just traversed across a number of displays, to suddenly lose track of their intent or goal. P5, for instance, when carrying out the control flow task stopped mid exploration and asked rhetorically “What was I doing here?”. P6 having switched concentration momentarily from the IDE to the task description, was unable to remember what he was doing having returned his focus to the IDE. A pattern was observed. The participant became distracted, in some cases due to cognitive drain related to navigating between displays and in other cases when reconsidering the task description or talking to the

facilitator, resulting in a loss of task focus (perhaps the participants short term goal being pushed out of working or short term memory). P1 said that when using the standard interface he tended to ‘miss things and forget information’.

In response to this occurrence, what is referred to here as task disorientation, a general pattern of re-orientation was observed in which the participant would navigate back over their previous locations using the back button or editor tabs in an effort to regain context. Essentially reviewing their previously visited locations in an attempt to remind themselves of their intent. If context could not be regained in this manner, the participant would restart exploration from a known location, generally the initial location specified in the task description, and begin to recreate their intent from that point.

The data also highlights this general trend. Participants carried out a greater deal of backtracking on the standard interface, 10x greater (See Chapter 7, Section 3) and also took longer to perform their tasks, 14% longer (See Chapter 7, Section 1). Participants also carried out a significantly greater amount of navigational actions, 31% more than the inline interface. These units of data, along with the observations, suggest that participants had difficulties maintaing navigation context when using the standard interface, and consequently performed a significant amount of backtracking/navigational actions in order to reorient, a process which is time consuming. The satisfaction questionnaires also reflect this conclusion with participants reporting that the code was harder to navigate using the standard interface and that they more often got lost and confused when exploring the code.

7.7.1.2 Revisiting known locations

Participants were observed to suffer from disorientation when unable to find locations and elements which had previously been encountered during the exploration tasks. Many of the tasks in the experiment naturally required the participant to explore from a given source code location, and then return to follow an alternate exploration route, or simply refer back to previously visited code as part of the comprehension process.

It was observed that participants had issues finding and returning to particular locations or elements in the system, which often resulted in frustration and in some cases progressed to disorientation. A number of aspects to this phenomenon were identified:

- Problems with the history stack
- The homogenous nature of source code
- A lack of code familiarity

The history stack in modern IDEs such as Eclipse is essentially a one dimensional list which allows the programmer to navigate back and forward over a sequential record of previously visited locations (See Chapter 3, Section 3.2.7). However, when a programmer is exploring source code they will often backtrack to a previous location and pursue an alternate route through the code resulting in a branch or digression in navigation history. The linear history stack cannot represent a digression in addition to the original path, and will consequently discard forward history. This results to a situation where a programmer may return from a particular location, then navigate to a new

location, and will ultimately be unable to return to the original location via the history stack. This situation was observed a number of times over the course of the experiment. The participant backtracked, intending to return to an element of interest but ended up in an unexpected area of the code resulting in distraction and over concentration on interface manipulation.

Another reason for participants being unable to locate known information was the homogenous nature of the source code. Participants sometimes simply didn't recognize the code they were looking for and bypassed it when backtracking on the history stack or flipping between the editor tabs. This issue seemed to be exacerbated during the control flow and inheritance tasks where participants were navigating through a series of large method bodies where the method name and inter-construct spacing and formatting were sometimes not fully visible in the editor viewport.

The final aspect of the phenomenon was a lack of familiarity with the source code structure. Essentially, participants were unable to find particular locations and elements because they did not have a clear idea of the structural organization of the system. For instance, while an experienced programmer might be expected to know the file or type within which a particular program element of interest is located, and thus easily find it in the package explorer, the participants, in general, had not built up this level of familiarity and instead searched for elements in the display space which was exhaustive and problematic.

The outcome of being unable to locate a location or program element in the system was generally a case of distraction which in some cases exacerbated into a loss of

task focus and disorientation. For instance, during the control flow task P6 followed a number of digressions and upon returning to the original branch point, after some frustration caused by flipping between the editor tabs, was unable to recall what he had intended to examine next. He ended up restarting exploration from the initial location specified in the task description.

7.7.1.3 Task context

Over the course of the experiment, it was observed that a number of participants used the visible set of editor tabs as a form of rudimentary task context. This representation was leveraged as a navigational and orientation aid, and in some cases even as a conceptual map during subsequent exploration activities.

Four participants p0, p1, p2 and p3 were all, at one point observed to consistently use the set of visible editor tabs for reasoning and navigation between program elements related to an exploration task. This was not a systematic behavior but seemed to occur due to the participant becoming disorientated as a result of a consistent inability to find particular pieces of code in the display space or comprehend casual relationships between fragmented source code elements. These participants were essentially avoiding disorientation, a commonly observed trend during information exploration activities (Henderson & Card 1986; Watts-Perotti & Woods 1999; De Alwis & Murphy 2006). Without an explicit representation of the elements associated with their task in which to reason about and orient, they adapted a rudimentary system using the editor tabs. This theory was given strength by the fact that many of the participants would close all open

editor instances at the end of a particular exploration task, in effect clearing their context in anticipation of the upcoming task.

It was also noticed that editor tabs were used as a conceptual aid. P2 arranged his editor tabs to mirror the control flow over a series of source code documents. This served as a sequential navigation aid in addition to a high level overview of the control flow structure.

7.7.1.4 Display thrashing

Thrashing was a common theme throughout the experiment, more so than was expected considering related work on the topic (De Alwis & Murphy 2006). All of the participants were observed, at one point or another, to thrash between source code displays. The thrashing was generally related to synthesizing an overview of fragmented source code in order to aid the comprehension process. P1 mentioned that it was 'difficult to trace control flow through a complex hierarchy' when observed to thrash back and forth between a number of sequential displays on the polymorphic task.

P2 was the only participant who was observed to use multiple editors in a simultaneous manner and thus avoid thrashing between related displays. This occurred on two occasions . However, the approach resulted in some frustration, P2 had some issues managing horizontal space and was unable to adjust the editor so as to fit all of the two source code locations in the single editor area. There was a considerable amount of interface adjustment carried out to this end. Furthermore, P2 seemed to forget which editor he was to focus on after conversing briefly with the facilitator, sometimes staring at

the screen for a moment. P2 also became annoyed when a declaration opened from the first visible editor replaced the code contained in the second editor thus breaking his carefully arranged interface layout.

Although the declaration view was open by default and its functionality described in the introduction to the experiment session, no participants were observed using it or taking further interest in its utility.

Thrashing is also suggested in the data. Forward and backward navigation was much more pronounced on the standard interface, as was the overall amount of navigational actions or display switches carried out (See Chapter 7 Sections 1 and 2).

7.7.2 Disorientation observed using the inline interface

Having gained an insight into the incidence of disorientation experienced on the standard interface, focus shifted to understanding the effectiveness of the inline interface at alleviating disorientation. Fundamentally, the aim was to ascertain if disorientation was reduced over the task sets, and if so, how and in what particular circumstances. It was also desirable to determine if the inline interface resulted in additional disorientation (unique to the inline mechanism itself) or problematic usability issues - in terms of both the core concept and the specific implementation provided by the fluid source code editor prototype.

7.7.2.1 Visible representation of navigation history/context

The study observations suggested that the visible representation of navigation history supported by the inline interface was reasonably effective at reducing disorientation associated with the lack of navigation context in the IDE. Two aspects were observed:

- A reduction in cognitive drain related to loss of context/display switching
- Ease of re-contextualization

On the inline interface, participants were able to carry out the majority of their exploration using inline introduction as opposed to explicitly navigating between discrete source code locations necessary on the standard interface. The inline interface resulted in a substantial reduction - 89% on average - in display switching (involving a total replacement of content) over the task sets. Consequently, it was observed that participants were significantly less prone to becoming distracted by the exploration process itself. This improvement in focus may have been due to the visible navigation context easing the cognitive requirement on the programmer to maintain context and orientation in the code space, in addition to elimination of the distraction associated with continual transition between discrete source code displays and information contexts.

When participants did get distracted, primarily due to clarifying some aspect of the task description or conversing with the facilitator, they were, in most cases, able to re-contextualize based on the visibly introduced source code declarations. P0 having returned from re-reading the task description commented that the inline interface

provides a ‘nice summary of explored elements’ before continuing his task. P1 mentioned that ‘the control flow path was easy to overview’ and P3 stated that the inline interface made it ‘easier to keep track of locations’ and that he liked how the programmer ‘stayed in the single view’. There was no observed incidents on the inline interface where a participant felt the need to backtrack through visited code in order to regain a sense of their current intent and focus. This thesis proposes that these observations account for a significant fraction of the reduction in task completion times and reduced use of the history stack recorded during the inline interface tasks.

However, beyond these positive findings, a problematic tendency associated with the inline interface with respect to the visible representation of navigation context was identified. Once a certain saturation of introduced information had been achieved (generally 6 or more levels in an inline expansion tree or when a particularly sizable declaration was introduced which required scrolling of the introduced content) participants began to show signs of losing orientation in the ‘expansion space’. On a number of occasions, participants became confused when considering code deeply embedded in a large expansion tree. P3 indicated that he became ‘lost’ when a large amount of source code was introduced inline into a single view. When asked to elaborate, P3 said that there was ‘too much going on’ on the screen. A number of reorientation strategies were also observed. Either the participant would give up the effort of tracing the various levels and close the expansion tree to restart exploration, or the participant would navigate explicitly to the most recently expanded source code declaration,

continuing exploration from that location, in essence moving to a fresh expansion context.

7.7.2.2 Exploratory digressions

On the standard interface, it was observed that participants experiencing issues when attempting to return to previously visited locations in the code space having pursued an exploratory digression.

Using the inline interface, it was noticed that participants rarely utilized the standard history stack provided by the IDE, preferring instead to leverage the visible representation of navigation history provided by the fluid editor. Because the fluid editor supports digressions in context - the programmer can evaluate a digression without explicitly leaving the original context- it was observed that the act of retuning from a digression was, in general cognitively effortless, simply a matter of collapsing the appropriate level in the expansion tree. P6 mentioned during exploration on the control flow task that it was easy to ‘back out of wrong turns’ because he could ‘see the exploration path’.

A portion of the improved task completion times (See Chapter 7, Section 1) was attributable to the ability to pursue exploratory digressions without the requirement to invest in cognitively expensive navigation away from the core context and potentially exhaustive display searching in order to return to the original location. These observations account for the results indicating that participants performed a lesser amount

of navigation actions on the inline interface (average of 31% reduction over the task sets) (See Chapter 7, Section 2).

7.7.2.3 Comprehending fragmented code

In terms of comprehending fragmented source code, the inline interface presented mixed results. Up to a certain level of introduction the inline interface seemed to help participants comprehend fragmented code, particularly when there was a need to examine multiple fragments of source code from the same expansion level in a simultaneous manner or when comprehending scattered control flow involving small to medium scale method declarations. P6 during the control flow task described an advantage of the inline interface as simply ‘seeing everything at once versus paging through’. It was observed that the ability to introduce search results in an inline manner was also useful, particularly the ability to examine multiple results in a simultaneous manner,. Numerous participants commented on the usefulness of this feature .

However, the issue of saturation and overly large declarations was again problematic. Once a certain saturation point was reached (expansion tree complexity or size), participants would lose orientation in the expansion space. Some participants were observed attempting to trace the various levels of the expansion tree with limited success. At one point during the experiment P6 seemed to become disoriented when considering the expansion tree associated with the control flow task. A large amount of nested inline information was visible on the screen and P6 exhibited a degree of confusion when tracing execution flow through levels. In response, P6 closed the main sub tree and

restarted inline exploration. It seemed that the act of creating the tree was an important part of the comprehension process. When asked to expand on the problem, P6 said that ‘there was too much code on the screen’ and that ‘it was difficult to see how the pieces fit together’. The introduction of large method and type declarations was also a considerable problem - causing surrounding context to be pushed out of the visible viewport and requiring the participant to scroll the display to view the introduced code.

A significant aspect of the comprehension and orientation problem in the expansion space was the the interline expansion technique - whereby the source code is spilt at the annotation anchor. The splitting of source code declarations seemed to be an issue in terms of participants tracing and comprehending introduced code. Perhaps a more promising approach was the original inline interface design where the declaration was introduced inline after the the anchor declaration, thus avoiding the need to split individual declarations.

It was observed that horizontal screen real-estate also became an issue in terms of large expansion trees. Once an expansion tree reaches a certain size, the accumulation of indentation (part of the differentiation mechanism) lead to the need for horizontal scrolling which required interface adjustment and exacerbated traceability issues between the various levels of nested introduction.

7.7.2.4 Miscellaneous user experience concerns

It was observed that fluid annotations did not interfere with the reading and exploration of source code which was positive. None of the participants complained about fluid

annotations obstructing comprehension of code and during the exit interview P1 mentioned that annotations ‘don’t get in your way’. When asked to agree or disagree with the statement ‘The fluid annotations were distracting...’ only one participant agreed, 7/9, and the average score associated with the statement was 2.3/9 (0 referring to total disagreement).

However, a number of issues regarding fluid annotation placement were observed during the experiment. The first issue related to the placement of fluid annotations was associated with method references. The existing design places the annotation on the closing parameter bracket as opposed to directly after the reference word such as in the case of types, fields and variables. In the presence of nested method references (one method reference acting as a parameter to another), it was observed that participants had difficulty identifying the particular annotation associated with the ‘inner’ and ‘outer’ reference. Furthermore in the presence of long or multi line parameter specifications, in which case the fluid annotation may be located at a significant distance from the actual method reference, participants were observed to have problems finding the annotation associated with the reference. Perhaps a more promising approach would have been to associate the annotation with the word of source code itself or use a hyperlink style interaction mechanism (which is problematic due to the existing and ubiquitous use of hyper-links for explicit hypertext style navigation).

A second issue concerns the proximity of fluid annotations in a general sense. When a number of annotations are located in close proximity, as defined by the density of the code, bounding regions tend to overlap and thus participants were observed to have

difficulty selecting an annotation of interest. Again this issue could be largely rectified by providing a more proximate activation mechanism.

In terms of fluid annotation colour coding, the reaction from participants was mixed. P0 and P2 complained of colour blindness and thus indicated that the annotation colour model had limited usefulness. P6 described annotation colour coding as a disadvantage and P5 indicated that annotation colour code was not required. When asked to agree or disagree with the statement ‘The colour coding of the fluid annotations was helpful..’, the average score worked out as 5.1/9 (0 referring to total disagreement and 9, total agreement). Overall, participants seemed ambivalent to the colouring of fluid annotations.

The colour model associated with inline expansions gained greater favour. Participants related positively to the ability to visually distinguish between adjacent levels in the introduction hierarchy. P1 mentioned that it was ‘easy to differentiate between coloured code’. P1 also mentioned in the exit interview that he would like to see a visual differentiation between ‘normal’ code and ‘interfaces’ - a general extension of the idea.

7.8 Validity

Throughout the design, execution and interpretation of the experiment, maintaining validity was a primary and overarching concern. The aim was to produce an accurate, defensible and interesting set of results and findings. This section discusses the validity of

the experiment and describes the various attempts to balance and mitigate potential threats and bias.

7.8.1 Participants

It would have been ideal to carry out the experiment using only professional programmers with experience in both the Java programming language and the Eclipse IDE. However, due to practical considerations such as timing and availability of willing and appropriate people, the experiment was carried out using a set of participants whose experience with Eclipse and Java varied significantly. As such, the threat of 'novice effects' was presents. As was the need to distinguish between disorientation caused by a lack of familiarity with the IDE interface and the language, disorientation associated with interface design and navigation and comprehension of the source code.

To deal with novice effects each observed incidence of disorientation was interpreted using a predefined 'disorientation model' developed specifically for the project. The disorientation model (See Chapter 6) is essentially a set of heuristics (based on existing research in the field) which may be applied to a incident of disorientation to determine its probable nature and underlying cause. During the experiment sessions the facilitator noted all observed incidents of disorientation. Afterwards the video and screen captures were synchronized and each recorded incident of disorientation was scrutinized in terms of the disorientation model. Using this process those incidents of disorientation which were overly influenced by novice effects were discarded.

The study results results (completion time, navigation actions etc.) were probably somewhat skewed by novice effects. However, as broad patterns in the data are primarily of interest this was considered an acceptable situation.

7.8.2 Tasks

The initial plan for the experiment was to have participants carry out live maintenance on a system. Essentially, each participant was to be given a set of small scale maintenance tasks, and would then be required to explore the system, identify and comprehend the appropriate code, and make the necessary alterations. This design represented the most valid approximation of real life programmer behavior, beyond an observation oriented field study.

However, during the study pilot it was observed that the participants, having varying degrees of experience and ability, found it very difficult to complete their maintenance tasks in a realistic period of time. Initially, the idea of reducing task size and complexity to boost completion times was considered. However, as tasks were simplified the realized dawned that participants would not carry out adequate source code exploration (both amount and type) for the study to gather sufficient data and identify patterns. Eventually, it was decided to have participants carry out pure exploration tasks and abandoned the idea of live maintenance.

There are two treats to validity associated with the task selection. The first threat is concerned with realism, and the second scope. Programmers generally carry out source code exploration in relation to specific development and maintenance tasks (Singer *et al.*

97). As such the tasks, in which a programmer explores source code in order to answer a number of predefined questions, lacks realism. Secondly, the task selection was designed to exercise the available functionality of the fluid source code editor. As such it could be argued that the task definitions were biased towards the inline interface.

Chapter 8

Conclusion & Future work

"*What are the advantages and disadvantages of inline source code exploration, and what effect does it have on programmer disorientation?"*

- Research question.

This thesis has explored the concept of inline source code exploration as a means of reducing the incidence of programmer disorientation during source code exploration activities in an IDE setting.

The methodology involved the development of a prototype implementation of inline source code exploration for the Eclipse IDE, entitled the 'fluid source code editor'(Desmond *et al.* 2006). The fluid editor supports the inline exploration of Java source code. This allows the programmer to explore code by introducing related declarations and search results into the context of a focal source code display, in an interactive, progressive and nested manner. The approach contrasts with the traditional

mechanism of explicitly navigating between discrete isolated displays. The prototype also supports to the introduction of non source code artifacts such as images and web resources.

The fluid editor was used as the basis of a user experiment designed to compare the level of disorientation experienced over a series of predefined source code exploration tasks. Participants completed half the tasks using an inline exploration interface (the fluid editor), and the other half using a standard exploration interface without inline capabilities (the standard Eclipse IDE). Disorientation was measured using a combination of metrics such as task completion time, visual momentum in the interface and navigation activity, in addition to an in depth analysis of participant behaviour, comments and gestures observed over the course of the experiment. Observations of disorientation were subsequently interpreted using recognized and accepted patterns of disorientation mined from existing research literature, both in the field of general and programmer specific disorientation.

The findings of the experiment suggest that participants using the inline interface experienced a reduced incidence of disorientation and consequently enjoyed increased productivity when performing the exploration tasks.

8.1 Trends

The experiment was exploratory in nature, focusing on trends rather than a fixed hypothesis. Participants experienced a 14% reduction in task completion times using the

inline interface, a 31% reduction in overall navigation carried out and an 89% reduction in cognitively expensive switching between discrete source code displays.

Participants expressed greater satisfaction with the inline interface in a number of categories including ease of use, perceived confusion and spatial awareness in the code space (based on satisfaction questionnaires filled out by participants after each task set). Participants also commented favourably on the inline source code exploration approach during feedback interviews and all agreed that they could readily envisage themselves making use of the technology if made available as a mainstream IDE based offering.

8.2 Findings

The findings of the research fall into three categories. Findings associated with programmer disorientation in the standard IDE. Findings related to the effectiveness of the inline interface at alleviating identified programmer disorientation, and related to the usability of the approach. And finally, findings regarding the usability of the fluid source code editor prototype.

8.2.1 Disorientation in the standard IDE

Based on observations gathered during the experiment, specifically on the standard interface (without inline capabilities), a set of factors and situations which typified the incidence of programmer disorientation occurring during source code exploration

activities in the IDE environment was synthesized. These findings are summarized for quick reference in Table *8.1*.

The cognitive drain associated with explicit navigation between sequences of isolated source code displays can result in distraction and loss of task focus.
A lack of navigation history/context in the IDE can lead to difficulties re-contextualizing after a loss of task focus and also increases cognitive overhead during exploration tasks.
Programmers have issues finding known locations in the IDE display space due to: • The limited history support in the IDE (inability to record digressions) • The visually homogeneous nature of source code (Makes it difficult to identify code in the IDE display space) • A lack of familiarity with the source code structure forces the programmer to search within the display space
Thrashing is common when attempting to understand and conceptually model fragmented source code. The IDE support for viewing multiple source code displays is limited and/or too cognitively expensive.

Table 8.1: A summary of findings related to the incidence of programmer disorientation during source code exploration activities in the Eclipse IDE.

It was noticed that, in accordance with existing research (Zelwegger *et al.* 2000), the process of explicitly navigating between discrete source code displays, a fundamental aspect of modern IDEs exhibiting a keyhole display architecture with low visual momentum, is a cognitively draining process which results in problems maintaining focus during exploration activities. Furthermore, because there is no explicit representation of navigation context (the path or sequence of source code elements/locations leading to the

currently visible source code display), programmers experience difficulties regaining orientation and focus having become distracted in the interface. This finding coincides with the general idea that navigation context is an important element in reminding a programmer of their ongoing intent and maintaining orientation in code space (Thüring *et al.* 95; Storey *et al.* 1999; De Alwis & Murphy 2006).

It was noticed that finding and revisiting previously visited source code elements and locations was problematic and occasionally resulted in disorientation due to excessive concentration on display searching and interface manipulation activities. A number of discrete factors underpinning with this phenomenon were identified. The history stack in modern IDEs such as Eclipse is limited to a single linear path of visited locations and consequently cannot record exploratory digressions. This limitation leads to difficulties returning from digressed code. The visually homogenous nature of source code results in problems identifying relevant code in the IDE display space. Finally, a lack of familiarity with the organization structure of the code forces programmers to rely on exhaustive searching in the display space, as opposed to cognitively cheaper facilities such as direct navigation via the package explorer or indexing based on type or resource name. Exhaustive searching in the display space is cognitively demanding and can lead to disorientation.

Programmers suffer from a lack of task context in the IDE (a summary of the source code artifacts related to a particular exploration task) and will sometimes adapt an ad-hoc representation using editor tabs. This rudimentary representation is used as an orientation and navigation aid, and in some cases as a conceptual map of the exploration

space. It was observed that the creation and use of ad-hoc task context was not systematic but occurred as a response to perceived disorientation in the interface, generally related to finding information in the display space and comprehending complex interactions between source code elements.

Finally the study indicated that thrashing is a common activity during the exploration and comprehension of fragmented source code, far more that was expected. Moreover the facilities available in modern IDEs to support the simultaneous consideration of related source code, which could mitigate the problem (such as multiple editor displays, pop-ups and the declaration view in Eclipse) are generally too limited or cognitively expensive to use. For instance comparing two source code displays in a single editor requires considerable interface adjustment to the point where the programmer may lose focus on their task.

8.2.2 Inline source code exploration

The study indicated that the inline interface (the fluid source code editor) was successful at alleviating certain aspects of the disorientation problems identified on the standard interface. The effectiveness of the inline source code exploration approach is summarized for quick reference in Table *8.2*.

Inline source code exploration eliminates the cognitive drain associated with explicit navigation between source code displays. As a result, programmers are less prone to losing task focus.
Inline source code exploration provides a visible representation of navigation history/ context. This representation eases re-contextualization after distraction, supports orientation in the code space and reduces the programmers cognitive overhead.
Support for exploratory digressions in context eases the programmers return from an exploratory digression and in many cases eliminates the need to search for previously visited code.
Inline source code exploration supports the exploration and comprehension of fragmented source code by providing a visible representation of navigation history and allowing the programmer to view multiple code fragments with minimal interface manipulation

Table 8.2: A summary of findings related to inline source code exploration, specifically its effectiveness at alleviating programmer disorientation.

In general the study indicated that the ability to explore source code by progressively introducing related source code declarations into a single source code display (the core tenet of the inline approach) reduces the cognitive drain on the programmer during exploration activities. It was observed that participants were far less prone to becoming distracted and losing task focus when introducing source code as opposed to explicitly navigating between discrete source code displays. This phenomenon is a result of the visible navigation context/history relieving the programmer of the cognitive burden required to maintaining orientation in the code space, as well as providing an explicit visual reminder of ongoing intent.

When a programmer does become distracted, a visual representation of navigation history supports re-contextualization. It was observed during the study that when participants become distracted on the inline interface, generally when considering the descriptive prompts associated with the exploration tasks or conversing with the experiment facilitator, re-contextualizing to the ongoing exploration focus was facilitated by the visible navigation summary provided by the inline interface. On the standard interface, without a visible representation of navigation history, participants would occasionally need to backtrack to previously visited locations in order to regain navigation context.

The ability to pursue and evaluate exploratory digressions without leaving the original exploration context reduced the requirement to search through the display space in order to return to previously locations in the exploration path, an activity which is prone to disorientation. It was observed that when using the inline interface participants were able to return from a pursued digression in a cognitively cheap fashion, simply by closing the appropriate level in an expansion tree.

The exploration and comprehension of fragmented source code was somewhat aided by the inline interface. Participants were able to get an overview of fragmented source code via the navigation history provided by the inline interface. This facility was particularly useful when there was a need to examine multiple fragments of source code from the same expansion level in a simultaneous manner, or when comprehending scattered control flow involving small to medium scale method declarations.

8.2.3 Disorientation in expansion space

In addition to observing the effectiveness of inline source code exploration at alleviating disorientation, a type of disorientation specific to inline exploration was also identified. This phenomenon is referred to as ‘disorientation in expansion space’. Essentially once a certain saturation of introduced information is achieved (generally 6 or more levels of nested inline introduction, or when a particularly sizable declaration is introduced which requires scrolling of the introduced content) programmers begin to show signs of losing orientation in the introduced source code.

On a number of occasions throughout the experiment, participants suddenly lost track of their location in the introduced source code, and consequently suffered from a loss of intent or focus. The same issue affected the comprehensibly of introduced source code, participants has issues comprehending complex expansion trees as a result of being unable to track relationships between introduced source code declarations. A significant factor contributing to the incidence of disorientation in expansion space is the introduction mechanism used by the fluid source code editor. The fluid editor will split source code declarations in order to introduce nested declarations in an inline manner. However when a number of declarations are split in a nested manner it becomes difficult to identify and trace the sequence of introductions leading to the currently focal introduction. Essentially, the traceability of the code is compromised.

A potential solution to the issue of disorientation in expansion space is the use of an after declaration introduction technique whereby the introduced declaration is located after the source declaration. This would eliminate the need to split declarations and thus

maintain the traceability of the code. However, the approach complicates the existence of multiple declarations originating from a single declaration. Multiple declarations would need to be stacked in the order of their introduction. The exploration of introduction and interface design techniques to combat disorientation in expansion space is a significant item of future work.

8.3 Future work

This research effort, focusing on inline source code exploration and programmer disorientation, has opened up a number of valuable research and development avenues which are deserving of additional consideration.

8.3.1 Inline editing

Inline editing is a compelling concept which has been raised at various points, by various people, throughout the research effort associated with this thesis. Having demonstrated or spoken about the fluid editor prototype, an invariable request is for the ability to edit introduced source code declarations. Because the primary focus if this research was source code exploration, this particular avenue of investigation has never been pursued beyond discussion and some playful prototyping.

An interesting element of future work would be to look into the concept of editing inline declarations and determine if and how this functionality might be useful for programmers working with source code.

8.3.2 Further evaluation

The user study associated with this research was limited both in terms of its size and its scope. Only eight participants were involved, and more significantly, the experiment involved relatively small and limited exploration tasks. To really get a sense for the usability of inline source code exploration it would be necessary to perform an expanded user experiment, including a larger participant base and a more realistic task set.

The ideal situation would be a field study in which professional programmers could be observed performing their normal development and maintain tasks in an IDE with the fluid editor installed.

8.3.3 Improvements to existing work

The fluid source code editor is a rudimentary but stable implementation of inline source code exploration for the Eclipse IDE. However based on the user experiment some areas of the system, and the overall approach, need additional prototyping and development:

- Design and placement of fluid annotations
- Introduction technique and format

8.3.3.1 Design and placement of fluid annotations

It was observed during the study that participants experienced issues identifying the appropriate fluid annotation for a given source code reference. This was particularly the case for method references where the annotation is placed at the closing parameter bracket. It was sometimes difficult for users to identify the correct annotation in a complex block of nested source code. Furthermore, when a large number of annotations are placed in close proximity it proved difficult to select one individual annotation due to overlapping bounding regions. The design and placement of fluid annotation needs some additional design and evaluation.

8.3.3.2 Introduction techniques

The introduction technique implemented by the fluid editor needs to be reconsidered in light of newly developed knowledge related to ‘disorientation in the expansion space’, See Chapter 7, Section 7.2. The splitting of source code declarations as part of inline exploration is problematic as it often leads to highly complex expansion trees. As a result the programmer has difficulty tracing the relationships between nested inline declarations.

The size of introduced declarations is also a significant problem. Some source code declarations, generally methods and types, can be larger than the available editor viewport. When they are introduced, the programmer may need to scroll out of the existing context to view all of the introduced code. This introduces scope for

disorientation associated with inline source code exploration. Some additional work should focus on preventing the introduction of large declarations, or more ambitiously, displaying large declarations in such a way that they are readable and explorable, but do not take up all of the available display space.

8.4 Vision

Having spent a number of years immersed in the field of programmer disorientation, I have developed a deep sense of the fundamental underlying problems which contribute to the phenomenon. I have also identified a significant void in the design of modern integrated development environments and how they support programmers during their work.

When carrying out a development or maintenance task on a particular software system, a programmer has three elements of knowledge which are of fundamental importance to maintaining focus and orientation (Storey *et al.* 1999):

- Navigational context
- Task context
- The emerging conceptual model

Navigational context refers to the programmers sense of spatial awareness in the code space, essentially where they are now, how they got to the current location and how to

find locations and elements of interest. Task context refers to knowledge of task being carried out, including specific artifacts in the code space which are of particular relevance. Finally, the emerging conceptual model is the programmers current understanding of the implementation, which is generally filtered on a particular aspect or feature of the system related to the current task.

At the moment, navigational context, task context and the emerging conceptual model are maintained at various levels of the programmers memory (working, short terms and long term). There is no explicit representation in the IDE that can be referred to or shared. This situation exhibits a number of drawbacks. Firstly, context represents a significant ongoing mental burden on the programmer. Secondly, when the programmer becomes distracted, either by external factors or interface problems, aspects of their mental representation of context may be forgotten. This process of losing context forms the essence of programmer disorientation, and the ongoing struggle to refresh context from the IDE display forms the essence of programmer re-orientation.

An extremely promising area of research in the field of programmer disorientation is the provision of a visible and interactive representation of the programmers navigation context, task context and emerging conceptual model in the IDE. This concept is referred to as providing a ‘prosthetic context’. A prosthetic context would significantly reduce the mental burden on a programmer, reduce the incidence of programmer disorientation and ease the process of re-contextualization in the event of external and internal distractions. Overall this would significantly improve programmer productivity and satisfaction.

I have carried out some Initial prototyping in the area of prosthetic context in the form of an Eclipse plug-in which allows the programmer to call up an overlay view containing a temporarily ordered graph of the current navigation context. The graph, which is implicitly computed from the programmers most recent navigation actions, presents a digression oriented view of the programmers navigation history with recently visited locations highlighted for orientation purposes. The programmer can leverage this display as an ongoing navigation and orientation aid during development and exploration tasks.

However, navigation history can also be leveraged to infer task context, and also form the basis of an explicit representation of the programmers conceptual model. For instance, one might extend the tool to apply a degree of interest model to the programmers navigation history (Kersten & Murphy 2005). This aim is to identify those elements in the system which the programmer repeatedly visits, and also determine visiting patterns such as how often the programmer visits one location after another etc. This information paints a picture of the programmers task context and indicates the 'implicit architecture' of the concern they are currently working. If stored and visualized in an appropriate manner this information can form the basis of an implicitly generated 'context model' associated with the programmers task.

Furthermore, if managed and packaged correctly a developer could even load a previous developer's 'context model' and use it to guide a related task. For instance a developer working on a bug associated with a particular feature could load and view the context model recorded by the original developer as they navigated between the various

program artefacts during the initial development effort. This idea adds an entirely new dimension to program documentation and modelling and programmer collaboration.

Chapter 8

Bibliography

Brooks, R. (1983). ‘Towards a theory of the comprehension of computer programs’. In *International Journal of Man-Machine Studies,* 18, pp. 543-554.

Brown, P.J. (1989). ‘Do we need maps to navigation round hypertext documents?’. In *Electronic publishing - Origination, Dissemination, and Design*, 2(2), pp. 99-100.

Bryant, A., Catton, A., De Volder, K. and Murphy G.C. (2002). ‘Explicit programming’. In *Proceedings of the 1st International Conference on Aspect Oriented software development*, pp. 10-18.

Card, S. K. and Nation, D. (2002). ‘Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface’. In *Proceedings of the Working Conference on Advanced Visual Interfaces*. pp 231-245.

Chin, J. P., Diehl, V. A. and Norman, K. L. (1988). ‘Development of an instrument measuring user satisfaction of the human-computer interface’. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 213 - 218.

Chu-Carrol, M.C, Wright, J. and Ying, A.T.T. (2003). `Visual separation of concerns through multidimensional program storage'. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pp. 188-197, New York, NY, USA. ACM.

Clark, H.H. and Haviland, S.E. (1974). ‘Psychological processes as linguistic explanation’. In D. Cohen, Ed., *Explaining Linguistic Phenomena*. Hemisphere, Washington, pp. 91–124.

Conklin, J. (1987). ‘Hypertext: An Introduction and Survey’, In *Computer*, 20(9), pp. 17-41.

Cook, R. I., and Woods, D. D. (1996). ‘Adapting to new technology in the operating room’. In *Human Factors*, 38, pp. 593–613.

De Alwis, B. and 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.

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

Dijkstra, E. W. (1968). `Letters to the editor: go to statement considered harmful'. In *Communications of the ACM*, 11(3), pp. 147-148.

Eclipse (2009a). ‘Eclipse.org home’, [Online] available: http://www.eclipse.org [accessed May 01, 2009].

Eclipse (2009b). ‘RFC Loss of Context - Draft 1’, [Online] available: http://dev.eclipse.org/viewcvs/index.cgi/platform-ui-home/loss-of-context/Proposal.html?view=co [accessed May 01, 2009].

Eclipse (2009c). ‘Eclipse Java development tools (JDT)’, [Online] available: http://www.eclipse.org/jdt [accessed May 01, 2009].

Eclipse (2009e). ‘Eclipse Mylyn Open Source Project’, [Online] available: http://www.eclipse.org/mylyn [accessed May 01, 2009].

Eclipse (2009f). 'Folding in Eclipse Text Editors', [Online] available: http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html [accessed May 01, 2009].

Eclipse (2009g). 'SWT: The Standard Widget Toolkit', [Online] available: http://www.eclipse.org/swt [accessed May 01, 2009].

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

Elm, W.C. and Woods, D.D. (1985). 'Getting lost: A case study in interface design'. In *Proceedings of the 29th Annual Meeting of the Human Factors Society*, pp. 927-931.

Elves, R.D. (2005). 'NavTracks - Helping Developers Navigate Source Code'. MSc. Thesis, University of Victoria, Victoria BC.

Foss, C.L. (1989a). 'Tools for reading and browsing hypertext'. In *Information Processing and Management: an International Journal*, 25(4), pp. 407-418.

Foss, C.L. (1989b). 'Detecting users lost: empirical studies on browsing hypertext', In Technical Report (Rapports de recherche) No 972, INRIA, Sophia-Antipolis.

Garrett, L.N., Smith, K.E. and Meyrowitz, N. (1986). 'Intermedia: issues, stratagies, and tactics in the design of a hypermedia document system'. In *Proceedings of the 1986 ACM conference on Computer-supported co-operative work*, pp. 163-174.

Gold, R., Chang, B.W., Zellweger, P.T., and Mackinlay, J (2000). 'Fluid Fiction: Harry the Ape'. *Exhibit at the San Jose Tech Museum of Innovation*, March 1 - September 7, 2000.

Halasz, F.G. (1988). 'Reflections on NoteCards: seven issues for the next generation of hypermedia systems'. In *Communications of the ACM*, 31 (7), pp. 836–852.

Halasz, F.G., Moran, T.P. and Trigg, RH. (1986). 'Notecards in a nutshell'. In *ACM SIGCHI Bulletin*, 17 (SI), pp. 45-52.

Harrison, W. and Ossher, H. (1993). 'Subject-Oriented Programming - A Critique of Pure Objects'. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 411 - 428 .

Henderson, A. D. and Card, S. (1986). 'Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface'. In *ACM Transactions on Graphics*, 5(3), pp.211-243.

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

Igarashi, T., Mackinlay, J.D., Chang, B.W. and Zellweger, P.T. (1998). 'Fluid visualization of spreadsheet structures'. In *Proceedings of the IEEE Symposium on Visual Languages*, pp. 118.

Intellij (2009). 'Intellij IDEA', [Online] available: http://www.jetbrains.com/idea/ [accessed May 01, 2009].

JHotDraw.org (2009). 'JHotDraw start page', [Online] available: http://jhotdraw.org [accessed May 01, 2009].

Jakobsen, M.R. and Hornbaek, K. (2006). 'Evaluating a fisheye view of source code'. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 377 - 386.

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

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.J. and Irwin, J. (1997). ‘Aspect-oriented programming’. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp. 220-242.

Kim, H. and Hirtle, S. C. (1995). `Spatial metaphors and disorientation in hypertext browsing'. In *Behavior & Information Technology*, 14(4), pp. 239-250.

Ko, A.J., Aung, H. and Myers, B.A. (2005). `Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks'. In *Proceedings of the 27th international conference on Software engineering*, pp. 126-135, New York, NY, USA. ACM Press.

Mahmoud, E. G. A. (1993). Overcoming Disorientation in a Hypermedia Environent with a Visually Based Builder and Navigator Tool. PhD. Thesis, George Washington University.

Mantei, M.M. (1982). ‘Disorientation Behavior in Person-Computer Interaction’. Ph.D. Thesis, University of Southern California.

Mayes, T., Kibby, M. and Anderson, T. (1990). 'Learning about learning from hypertext'. In *Jonassen DH, Mandl H (Eds.), Designing hypermedia for learning*, pp. 227–250. London, UK: Springer-Verlag.

McAlesse, R. (1999). 'Navigation and browsing in hypertext'. In *Hypertext: theory into practice*, Intellect Books Exeter, UK, pp. 5-38.

Microsoft (2009). 'Visual Studio', [Online] available: http://www.microsoft.com/visualstudio/ [accessed May 01, 2009].

Murphy, G.C., Kersten, M. and Findlater, L. (2006). 'How Are Java Software Developers Using the Eclipse IDE?'. In *IEEE Software*, 23(4), pp. 76-83.

Nielsen, J. (1990). 'The Art of Navigating Through Hypertext'. In Communications of the ACM, 33(3), pp. 296-310.

Netbeans (2009). 'Welcome to netbeans', [Online] available: http://www.netbeans.org [accessed May 01, 2009].

Parnas, D. L. (1972). 'On the criteria to be used in decomposing systems into modules'. In *Communications of the ACM*, 15(12), pp. 1053-1058.

Pennington, N. (1987). ‘Stimulus structures and mental representations in expert comprehension of computer programs’. In *Cognitive Psychology*, 19, pp. 295-341.

Schneiderman, B. (1980). ‘Software Psychology: Human Factors in Computer and Information sciences’. Winthrop Publishers, Inc.

Schneiderman, B. and Mayer, R. (1979). ‘Syntactic/semantic interactions in programmer behavior: A model and experimental results’. In the *International Journal of Computer and Information Sciences,* 10(5), pp. 219-238.

Schümmer, T. (2001). ‘Lost and Found in Software Space'. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 9, Washington, DC, USA. IEEE Computer Society.

Singer, J., Elves, R. and Storey, M.A (2005). ‘Navtracks: Supporting Navigation in Software Maintenance’. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 325 - 334.

Singer, J., Lethbridge, T., Vinson, N. and Anquetil, N. (1997). ‘An examination of software engineering work practices'. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, pp. 21+. IBM Press.

Soloway, E. and Ehrlich, K. (1984). ‘Empirical studies of programming knowledge’. In *IEEE Transactions on Software Engineering,* 10(5), pp. 595-609.

Sourceforge.net (2009). ‘Fluid Editor Project’, [Online] available: http:// fluideditor.sourceforge.net [accessed May 01, 2009].

Storey, M.A.D., Fracchia, F.D. and Muller, H.A. (1999). `Cognitive design elements to support the construction of a mental model during software exploration'. In *Journal of Systems and Software*, 44(3), pp. 171-185.

Tarr, P., Ossher, H., Harrison W. and Sutton, S.M.Jr. (1999). `N degrees of separation: multi-dimensional separation of concerns'. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pp. 107-119, New York, NY, USA. ACM.

Thüring, M., Hannemann, J. and Haake, J.M. (1995). ‘Hypermedia and cognition: designing for comprehension’. In *Communications of the ACM*, 38(8), pp 57-66.

Van Dyke Parunak, D. (1989). `Hypermedia topologies and user navigation'. In *HYPERTEXT '89: Proceedings of the second annual ACM conference on Hypertext*, pp. 43-50, New York, NY, USA. ACM.

Von Maryhauser, A. and Vans, A.M. (1995). ‘Program comprehension during software maintenance and evolution’. In *IEEE Computer, pp 44-55.*

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

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

Woods, D. D. and Roth, E. M. (1988). ‘Cognitive systems engineering’. In *Handbook of human–computer cooperation*, pp. 3–41.

Woods, D. D. and Watts, J. C. (1997). `How Not to Have to Navigate Through Too Many Displays'. In *M. G. Helander, T. K. Landauer, & P. V. Prabhu (eds.), Handbook of Human-Computer Interaction*, chapter 26. Elsevier Science B.V., Amsterdam, second edition.

Yatim, N. (2002). ‘A Combination Measurement for Studying Disorientation’. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, 5(5), pp. 138.

Zellweger, P.T., Chang, B.W. and Mackinlay, J. (1998). 'Fluid links for informed and incremental link transitions'. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia : links, objects, time and space - structure in hypermedia systems*, pp 50-57.

Zellweger, P.T., Mangen, A. and Newman, P. (2002). 'Reading and writing Fluid hypertext narratives'. In *Proceedings of the thirteenth ACM conference on Hypertext and hypermedia*, pp 45-54.

Zellweger, P.T., Regli, S.H., Mackinlay, J.D. and Chang, B.W. (2000). 'The impact of Fluid Documents on reading and browsing: An observational study'. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp 249-256.

Appendix A

User Experiment questionnaires

Satisfaction Questionnaire

Standard Exploration Interface

Questions are presented on a scale from 0 to 9. Read each question carefully and then circle the number on the scale that most accurately represents your answer.

1. How did you find the code exploration interface in general?

Very poor 0 1 2 3 4 5 6 7 8 9 **Very good**

2.- 6. How was the interface to use?

Terrible 0 1 2 3 4 5 6 7 8 9 **Wonderful**

Hard 0 1 2 3 4 5 6 7 8 9 **Easy**

Frustrating 0 1 2 3 4 5 6 7 8 9 **Pleasant**

Boring 0 1 2 3 4 5 6 7 8 9 **Fun**

Confusing 0 1 2 3 4 5 6 7 8 9 **Clear**

7. It was clear, most of the time, where i was in the source code...

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

8. I often lost my orientation (got lost) in the source code...

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

9. I often felt confused when exploring the source code...

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

10. There was sometimes too much information on the screen at once...

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

11. How did you perceive the tasks?

Very poor 0 1 2 3 4 5 6 7 8 9 **Very good**

12. How would you rate your answers to the tasks?

Very poor 0 1 2 3 4 5 6 7 8 9 **Very good**

13. - 15. Was the source code...

Hard to understand 0 1 2 3 4 5 6 7 8 9 **Easy to understand**

Hard to overview 0 1 2 3 4 5 6 7 8 9 **Easy to overview**

Hard to navigate 0 1 2 3 4 5 6 7 8 9 **Easy to navigate**

16. Was information in the source code...

Hard to locate 0 1 2 3 4 5 6 7 8 9 **Easy to locate**

Satisfaction Questionnaire

Fluid Exploration Interface

Questions are presented on a scale from 0 to 9. Read each question carefully and then circle the number on the scale that most accurately represents your answer.

1. How did you find the fluid exploration interface in general?

Very poor 0 1 2 3 4 5 6 7 8 9 **Very good**

2.- 6. How was the fluid interface to use?

Terrible 0 1 2 3 4 5 6 7 8 9 **Wonderful**

Hard 0 1 2 3 4 5 6 7 8 9 **Easy**

Frustrating 0 1 2 3 4 5 6 7 8 9 **Pleasant**

Boring 0 1 2 3 4 5 6 7 8 9 **Fun**

Confusing 0 1 2 3 4 5 6 7 8 9 **Clear**

7. It was clear, most of the time, where i was in the source code.

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

8. I often lost my orientation (got lost) in the source code.

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

9. I often felt confused when exploring the source code.

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

10. Most of the time i had a good idea of the structure of the code.

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

11. There was sometimes too much information on the screen at once.

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

12. It was easy to determine the relationships between expanded pieces of code.

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

12. The fluid annotations were distracting...

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

13. The coloring coding of the fluid annotations was helpful..

I disagree 0 1 2 3 4 5 6 7 8 9 **I agree**

14. How did you perceive the tasks?

Very poor 0 1 2 3 4 5 6 7 8 9 **Very good**

15. How would you rate your answers to the tasks?

Very poor 0 1 2 3 4 5 6 7 8 9 **Very good**

16. - 18. Was the source code...

Hard to understand 0 1 2 3 4 5 6 7 8 9 **Easy to understand**

Hard to overview 0 1 2 3 4 5 6 7 8 9 **Easy to overview**

Hard to navigate 0 1 2 3 4 5 6 7 8 9 **Easy to navigate**

19. Was information in the source code...

Hard to locate 0 1 2 3 4 5 6 7 8 9 **Easy to locate**

Appendix B

User Experiment participant profile

Participant profile

Please read each question carefully and provide an answer in the space provided. Please write as clearly as possible keeping answers succinct.

Name: _____

Sex: (M/F) _____

Occupation:

_____
_____
_____
_____
_____
_____

Programming experience:

_____
_____
_____
_____
_____
_____

Eclipse experience:

_____
_____
_____
_____
_____
_____

Appendix C

User Experiment exit interview

Exit Interview

1. Did you prefer the fluid exploration interface or the standard exploration interface? Please elaborate.

_____
_____
_____
_____
_____

2. What advantages or disadvantages did you perceive when using the fluid exploration interface vs. the standard exploration interface?

_____
_____
_____
_____
_____

3. Can you identify any aspects of the fluid editor that stood out as particularly confusing or frustrating?

_____
_____
_____

4. Are there any features that you would like to see in future versions of the fluid editor?

Would you see yourself using the fluid editor or a similar system when programming?

Appendix D

User Experiment task descriptions

Task 1A

In the JHotDraw framework a DefaultSDIApplication instance handles the life cycle of a single document window presented in a JFrame. The JFrame provides all of the functionality needed to work with the document, such as a menu bar, tool bars and palette windows etc. A DefaultSDIApplication instance is used to host the JhotDraw sample application we looked at earlier.

Your task is to explore the init() method declared on the DefaultSDIApplication class and answer a set of simple questions. First you will need to locate the init() method in DefaultSDIApplication.java. DefaultSDIApplication.java is located in the org,jhotdraw.app package, you can find it with the package explorer view. When you have found the init() method and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Q1. On what interface is the init() method abstractly declared? Describe how the init() method is implemented in the type hierarchy from the abstract declaration down to the DefaultSDIApplication implementation.

Q2. The init() method declared on DefaultSDIApplication calls its superclass init() method, briefly describe the implementation of this superclass method. What is the type of the recentFiles field used in this method? What type is the applictionModel field used in this method?

Q3. The init() method declared on DefaultSDIApplication invokes the initLabels() method and the initApplicationActions() method in that order. In what class are each of these methods defined? Both the initLabels() and the initApplicationActions() methods use a common portion of code, can you identify this code?

Task 1B

In the JHotDraw framework a Drawing represents a container for figures. A Drawing organizes its Figures into a list and figures can be added and removed from a Drawing as needed. The DefaultDrawing class provides a basic implementation of the Drawing interface.

Your task is to explore the draw(Graphics2D) method contained in the DefaultDrawing class and answer a set of simple questions. First you will need to locate the draw(Graphics2D) method in DefaultDrawing.java. DefaultDrawing.java is located in the org,jhotdraw.draw package. When you have found the draw(Graphics2D) method and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Q1. The execution of the draw(Graphics2D) method is controlled by a lock object. Find the piece of code which instantiates the lock object. According to the Java doc associated with the declaration of the getLock() method, the lock object is designed to prevent what situation?

Q2. Briefly describe the implementation of the ensureSorted() method? After sorting the figures the draw(Graphics2D) method performs a clipping operation, culling those figures outside the clipping bounds based on the value of their drawing area. Compare the implementation of the getDrawingArea() method for the RectangleFigure and the EllipseFigure types and briefly describe the difference(s) and similarities.

Q3. The draw(Graphics2D) method on the DefaultDrawing class now calls the draw(Graphics2D,Collection<Figure>) method to draw the clipped figure collection. Can you identify any redundant execution in the draw(Graphics2D, Collection<Figure>) method considering the execution of the draw(Graphics2D) method?

Task 2A

The drawDrawing() method declared on the DefaultDrawingView class is responsible for drawing the figures contained in its associated Drawing object. But ultimately each figure is responsible for drawing itself.

Your task is to trace the control flow of the drawDrawing() method until execution reaches the drawFill() method on the EllipseFigure class. First you need to locate the drawDrawing() method in DefaultDrawingView class. DefaultDrawingView.java is located in the org,jhotdraw.draw package.

When you have found the drawDrawing() method and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

The line of code “drawing.draw(g)” is the root of the control flow hierarchy. Follow the execution of the this method until you encounter the drawFill() method on the EllipseFigure class.

Q1. Describe the execution hierarchy including details such as interfaces and looping constructs. Assume that the field drawing is of concrete type DefaultDrawing and any Figure objects encountered are instances of AbstractAttributedFigure.

Q2. Describe, at a high level, the steps involved in the draw operation? (use the the assumptions of variable type outlined above)

Task 2B

The DeleteAction class initiates the removal of a figure in the JhotDraw system The method actionPerformed(ActionEvent) is executed to initiate the removal of a selected figure or figures from the active Drawing.

Your task is to trace the control flow of the actionPerformed(ActionEvent) method of the DeleteAction class until execution reaches the removal of a figure from the active drawing object (a call to basicRemove(Figure)). First you need to locate the actionPerformed(ActionEvent) method in the DeleteAction class. DeleteAction.java is located in the org,jhotdraw.app.actions package. When you have found the actionPerformed(ActionEvent) method and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Follow the execution of the actionPerformed(ActionEvent) method until you find the piece of code which removes figures from the active drawing object (the first call to basicRemove(Figure)).

Q1. Describe the execution hierarchy including details such as interfaces and looping constructs.

Q2. Describe, at a high level, the steps involved in the draw operation?

Q3. Did you notice any potential issues with the code?

Task 3A

The findFigure(Point2D.Double) method declared on the Drawing interface is used to find the top level figure which contains a given point.

Your task is to find and explore the implementation(s) of the findFigure() method and answer a set of simple questions. First you need to locate the findFigure() method on the Drawing interface. Drawing.java is located in the org,jhotdraw.draw package. When you have found the findFigure() method and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Q1. List all types (concrete and abstract) which implement the findFigure() method of the Drawing interface.

Q2. Examine the various implementations of the findFigure() method and describe at a high level how they differ in terms of finding the figure for a the given point. What data structure(s) is used for the storage of figures in each implementation type?

Q3. The implementation of the findFigure() method in the QuadTreeDrawing and DefaultDrawing types both call a common method. Can you identify this method? Is this method duplicated in both types?

Task 3B

The addNotify() method declared on the Figure interface is called to inform a figure that it has been added to a specified drawing. The figure must then inform all registered FigureListeners that it has been added.

Your task is to find and explore the implementation(s) of the addNotify() method and answer a set of simple questions. First you need to locate the addNotify() method on the Figure interface. Figure.java is located in the org,jhotdraw.draw package. When you have found the addNotify() method and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Q1. List all types which implement the addNotify() method of the Figure interface (both abstract and concrete).

Q2. The implementation of the addNotify() method in the AbstractFigure class calls the fireFigureAdded() method. Examine this method and describe its execution flow. What concrete type is the listenerList field?

Q3. Compare the implementation of the addNotify() method in the AbstractFigure and the AbstractCompositeFigure classes. Briefly describe the differences at a high level.

Q4. What relationship does the AbstractCompositeFigure class have with the AbstractFigure class?

Q5. Now examine the implementation of the addNotify() method in the GraphicalCompositeFigure class. What processing is carried out? What relationship does the GraphicalCompositeFigure class have with the AbstractFigure class?

Task 4A

The TriangleFigure class represents a triangle in the JhotDraw drawing framework. Your task is to explore the implementation of the TriangleFigure class and answer a set of simple questions. First you need to locate the TriangleFigure class contained in TriangleFigure.java. TriangleFigure.java is located in the org,jhotdraw.draw package.

When you have found the TriangleFigure class and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Q1. At a high level describe how the findConnector(...) and findCompatableConnectior(...) methods differ from their superclass implementations. Describe how the findConnector(...) method is implemented in the type hierarchy from the abstract declaration down to the TriangleFigure implementation.

Q2. The createHandles(int) method on the TriangleFigure class is called to create a collection of handles (small adjustable widgets) used to manipulate aspects of the figure. On a call to the createHandles(int) method on a TriangleFigure instance a collection of handles is returned. List the various handle objects added to the returned handle collection in order of their addition to the return handle list. According to the Java doc on the createHandles(int) interface method, what is the significance of the detail parameter?

Q3. At a high level describe the difference between the setBounds(...) method on the TriangleFigure class and its superclass implementation? Can you recognize any potential issues with the superclass implementation?

Task 4B

The TextFigure class represents a visible piece of text in the JhotDraw drawing framework. Your task is to explore the implementation of the TextFigure class and answer a set of simple questions. First you need to locate the TextFigure class contained in TextFigure.java. TextFigure.java is located in the org,jhotdraw.draw package. When you have found the TextFigure class and are ready to being the task please select Tasks->Begin new task, enter the task name above and begin.

Q1. Consider the declaration of the getPreferedSize() method on the TextFigure class. Can you recognize any potential issues or anomalies with this method when compared with its superclass implementation? According to the Java doc on the interface declaration of the getPreferedSize() method what component uses getPreferedSize() to determine the preferred size of a figure.

Q2. The method read(DOMInput) is used to read the serialized state of a stored TextFigure object from an input stream. Compare the implementation of the read(DOMInput) method with its overridden superclass implementation. Describe in detail the essential difference(s) between the two implementations in terms of what gets read from the stream, the order of reading, and the use of the read data. Could you recommend a better implementation of the read(DOMInput) method for the TextFigure class?

Q3. The clone method declared on the TextFigure class is called to create a clone of the current TextFigure instance. Describe or sketch the execution flow of the clone() method up through the class hierarchy. What fields are assigned on the cloned object and what order are they assigned?

Appendix E

AOSD 2006 Poster

FLUID SOURCE CODE

FLUID DOCUMENT TECHNOLOGY IN ECLIPSE

MICHAEL.DESMOND@GMAIL.COM

TECHNOLOGY

Fluid Documents

Source code documents consist of primary information contained in the document along with explicit, implicit and semi implicit links to portions of related documents.

It is common that users examining a primary source document would like to examine related material without leaving the context of the primary document. Fluid documents provide optional, contextual access to supporting or linked information, allowing the reader to fluidly shift attention from primary to supplemental information and back again to the primary information.

Interactive source code

Source code is typically presented as static text within the document editor. An innovative feature of our fluid source code tools is the embedding of interactive, unobtrusive widgets directly into the source code presentation.

Code/Content folding

To reveal supplementary information at interactive points in fluid documents, code and content folding techniques are used. Both text based and widget based content can be unfolded within fluid documents.

FUTURE WORK

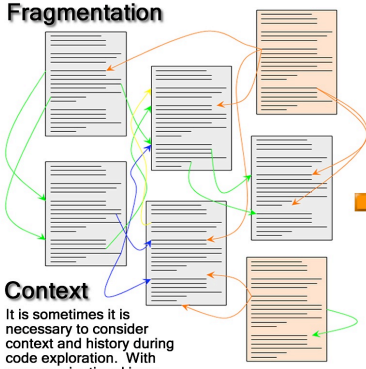
- Completion of the Eclipse fluid document tools and their release to the open source community.
- Develop further applications of fluid document technology in the software engineering field
  - Web development tools
  - Multi dimensional seperation of concerns editor
- Support tools to ease information overload and overview and navigate fluid documents
- Comperhensive user studies to ascertain the advantages of fluid source code views

APPLICATION

Fragmentation of software definition and its affect on Navigation and comperhension

Modern Integrated development environments, such as Eclipse, help programmers find, examine, edit and navigate between points of interest in code. IDEs represent source code in terms of static text based documents. Navigation occurs by "jumping" from positions in document to document and forces the programmer to synthesize information scattered across multiple points in multiple documents when comprehending code.

Fragmentation

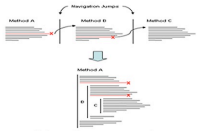


Context

It is sometimes it is necessary to consider context and history during code exploration. With every navigational jump, context is lost and the programmer may need to "jump back" to a previous point in code in order to refresh their memory of context.

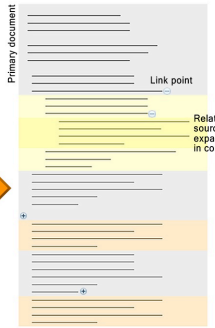
- Method invocations
- Aspect advice
- Polymorphic dispatch
- Inheritance relationships

Although the modular structure of code provides a large number of benefits in terms of software quality it is sometimes fitting that code be presented to the user in a way that temporarily breaks modularity in order to better facilitate examination, exploration and comprehension.

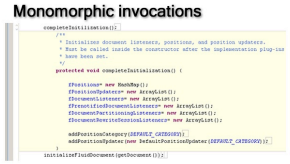


Fluid source code Views

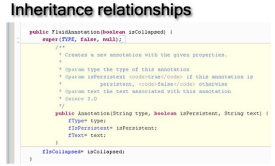
A fluid source code view is an enhancement of the traditional source code editor with support for fluidly exploring the software space related to the source file being edited. This exploration happens within the context of the primary source file being examined and allows the programmer to shift attention from the primary software element under examination to related elements without distraction.



Monomorphic invocations

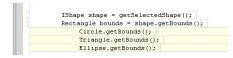


Inheritance relationships



Aspect oriented programming

Polymorphic/Abstract invocations



Goals

Reduce navigation through the software space
Reduce synthesization during code analysis
Support program comprehension patterns

michael.desmond@gmail.com

eclipse

OLLSCOIL LUIMNIGH UNIVERSITY OF LIMERICK

University of Victoria | British Columbia Canada

embarkinitiative