Remaining Oriented During Software Development Tasks: An Exploratory Field Study

Brian de Alwis and Gail C. Murphy

Department of Computer Science
University of British Columbia
Vancouver BC, Canada V6T 1Z4
{bsd, murphy}@cs.ubc.ca

TR-2005-23
July 17, 2005

Abstract

Humans have been observed to become *disoriented* when using menu or hypertext systems. Similar phenomena have been reported by software developers, often manifesting as a feeling of *lostness* while exploring a software system. To investigate this phenomena in the context of software development, we undertook a field study, observing eight developers of the open-source Eclipse project for two hours each as they conducted their normal development work. We also interviewed two other developers using the same tools but who were working on a closed-source system. The developers did report some instances of disorientation, but it was a rare occurrence; rather we observed strategies the developers used to remain *oriented*. Based on the study results, we hypothesize factors that contribute to disorientation during programming tasks as well as factors that contribute to remaining oriented. Our results can help encode best practices for code navigation, can help inform the development of tools, and can help in the further study of orientation and disorientation in software development.

1 Introduction

Software systems are complex, continuously evolving artefacts [1]. Before undertaking a change to any realistic software system, developers spend a large amount of time exploring and re-familiarizing themselves with the system to determine and understand the sections of the code relevant to that task [25, 26]. Integrated development environments (IDEs) include functionality to support this exploratory navigation, such as

---

search tools to follow semantically-meaningful relationships and navigation support features, such as web-style stack-based navigation.

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. These developers have noted that this problem of disorientation continues to occur even with modern IDEs such as the Eclipse Project's Java Development Tooling (JDT) [10], a widely-used open-source integrated development environment — despite a comprehensive effort to address suspected factors relating to complaints of disorientation [7]. The research literature also includes references to disorientation during software development tasks (e.g., [2, 15, 29, 30]), and some tools appeal to preservation of user context as an argument to the effectiveness of their techniques (e.g., [16, 20, 28]).

We believe an understanding of how disorientation occurs can lead to better ways to manage or prevent the disorientation, resulting in an improvement in the effectiveness of software developers. Recovering from disorientation takes time and effort and can result in information being forgotten or disregarded. In software systems, missing or faulty information can result in software errors [26].

To provide an initial characterization of disorientation as it occurs in software development, we conducted an observational field study of practicing software developers. This exploratory study sought answers to the following questions:

- How does disorientation manifest with respect to software development?
- Do expert developers become disoriented? If so, how often does disorientation occur?
- Are there factors of the development environment that exacerbate or prevent disorientation, or aid in recovery from disorientation?

We observed and interviewed eight developers of the Eclipse Project as they conducted their normal development work using the Eclipse JDT. We also interviewed two other IBM developers who used the Eclipse JDT as a development environment, but who were working on a closed-source system.[1] This kind of study allowed us to stress realism, an important consideration as we assumed disorientation was more likely to occur as developers work on larger systems in their actual work environment. This population was deliberately chosen to eliminate any possibility that disorientation may arise because of novice effects or unfamiliarity with the the environment. We chose to focus on developers using the Eclipse JDT as this development environment incorporates state-of-the-practice tools for navigation, programming and debugging for Java; none of the developers were responsible for portions relating to the Eclipse JDT.

While we observed some developers become disoriented, and had other developers report having been disoriented previously, we were somewhat surprised at the rarity of disorientation. Further analysis of the data collected indicates that developers evolve strategies to remain oriented, as well as use features of their development tools or other aids to maintain their context.

---

[1] Throughout this paper, we distinguish where necessary between the eight *Eclipse developers* and the two IBM *developers*.

We begin in Section 2 by reviewing background work in disorientation as it occurs in other fields and reports in software development. In Section 3 we describe the format and situation of the study. We describe the occurrences of disorientation found in the study in Section 4, and present an initial characterization of the types of disorientation found. In Section 5 and 6 we describe factors we hypothesize contribute to disorientation as well as strategies we observed developers using to remain oriented. Based on our findings, we make recommendations for both developers and IDE designers in Section 7. There are some potentially limiting factors to the study, which we discuss in Section 8. We finally conclude and discuss some possible future directions in Section 9.

Terminology. In seeking to clarify what disorientation means in software development we must be clear in our use of related terms. *Disorientation* (e.g., [30]), *loss of context* (e.g., [7]), and *getting lost* (e.g., [20]) are often used interchangeably to describe the psychological phenomenon of *lostness*, especially when describing phenomena experienced by software developers. In this paper we consider disorientation to be an encompassing concept.

2 Background

2.1 Disorientation in Other Domains

Disorientation has been studied in other computer-based domains, such as menu systems [13, 22], medical monitoring systems [35], hypertext (the *lost in hyperspace* phenomenon) [6, 14, 21], and spreadsheets [32]. Mantei [22], in the earliest study of disorientation, described disorientation as the self-awareness of being *mentally* lost (subjective lostness), as compared to — and independent of — being actually lost (objective lostness). Elm and Woods describe disorientation as when "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" [13, p.927].

Disorientation with computer-based systems occurs as the display presents only a very small window onto a very large information space, like peering through a keyhole [32]. This small aperture makes it difficult to locate information or to piece together a coherent picture of a situation, especially when the display system does not provide adequate support to the user to navigate within this information space. Watts-Perotti and Woods [32] describe that these navigation problems may manifest themselves in several ways: as feelings of lostness, where the users are unable to determine where they are or what they are examining; as display thrashing, where users must repeatedly switch between different displays to correlate information of interest; and as mental workload from interface management. Users develop work-arounds to prevent the occurrence of disorientation, and may even deliberately avoid the use of displays even though they provide additional information [32].

But disorientation may be induced by poor structure both in the interface (the medium) as well as the the organization of the information structure (the content) [22]. Information spaces structured as large, irregular networks are much likelier to induce disorientation; the introduction of some semblance of regular structure on the underlying network can dramatically change their navigability and aid in preventing disorientation [22, 24], But the user interface still plays an essential role in discovering that structure and preserving orientation [22]. Disorientation affects novice and expert users of the interface alike, and does not set in immediately [22].

The work on disorientation using hypertext considers both navigation and the user's intent. These problems observed have been grouped into three categories [21]:

navigational disorientation problems: where users are unable to access items of interest or identify their location, caused by incomplete or incorrect knowledge of the information structure;

task management problems: where users either fail to resume or complete previously-suspended tasks, or do not pursue planned digressions;

information management problems: where users are unable to synthesize or remember information found.

2.2 Disorientation in Software Development

Software development is somewhat different from these other domains. These other domains tend to allow creators to change the underlying organization of the information structure, and there is typically an inherent spatialization of the information structure. For example, hypertext authors can choose a linear document structure or a seemingly-random structure; spreadsheets have an inherent two-dimensional spatial structure. In contrast, software developers rarely have the opportunity to change the structure of their programming language software, and due to the sheer size and complexity of the semantic relations, there is no inherent spatialization [5].

The research literature includes several oblique references to disorientation during software development tasks. Several researchers have noted that browsing of class libraries can lead to disorientation [2, 15, 29, 30]. Janzen and De Volder [20] claim that developers become disoriented by switching between views, but provide no empirical evidence for this phenomenon. Storey et al. [28] describe several design elements for software visualization tools aimed to reduce disorientation. Some tools, such as AspectBrowser [16], SHriMP [27, 28], and and JQuery [20], appeal to preservation of user context as an argument for the effectiveness of their techniques, although their evidence only provides indirect substantiation of this claim.

We are not aware of any attempts to characterize the form of disorientation as it occurs during software development. Informal complaints from developers lead to the study we report here, to investigate what actually constitutes disorientation during software development.

2.3 Assessing Interfaces for Disorientation

Because software development tasks generally require correlating information gleaned from different *displays*, referring to the content displayed on a computer screen at some particular time, and given the important role played by user interface in discovering that information, we introduce a heuristic measure for evaluating the ability of a user interface to alleviate or prevent disorientation, named *visual momentum*.

Visual momentum is a measure of a user's ability to extract relevant information across views and displays [32, 33], and can serve as a proxy measure for evaluating a system's ability to convey the underlying network of an information structure. This concept was inspired by techniques used in cinematography to guide viewers where to look next while preserving their sense of orientation. Interfaces with high visual momentum aid in coordinating the information displayed, and users are able focus completely on their task and are not concerned with managing the interface. Interfaces with low momentum are essentially serial displays, where each display is perceptually independent of both its prior and subsequent displays, thus requiring the user to carry the mental burden of transitioning and reorienting between each display.

There are several techniques that increase the visual momentum for an interface [32, 33], which we list here. We will see instances of these in the study.

longshots: provide high-level overviews, which are generally tailored to a task;

functional overlays: display surrounding context of the display currently being examined;

navigational landmarks: identify prominent elements, helping users reorient between displays;

bookmarks: record positions of interest;

spatial dedication: support the establishment of a spatial frame of reference between the information displayed and its meaning.

3 Study Format

The study was structured as an observational field study of eight Eclipse developers (referred to as P1–P8) at IBM Canada's Ottawa Software Laboratory. The study was conducted in the two weeks prior to the penultimate milestone of Eclipse 3.0, named M8. Participants were observed as they pursued their normal development tasks using video taping, screen captures, and note-taking.

We also interviewed two former Eclipse developers (referred to as E1,E2) now developing a closed-source system using the Eclipse JDT as a development environment. Their project characteristics differed from the Eclipse developers in that they used a much larger number of components and subsystems in their work. We were unable to observe these participants at work due to confidentiality concerns.

The remainder of this section describes the study in more detail; limitations are left to Section 8.

3.1 Background of Participants

Participants were recommended for the study by their team leaders. Participation required having expertise in the Java programming language, experience in using the Eclipse JDT as a development environment, and to be currently performing development work.

All developers were male and between 25 and 40 years of age. All had used the Eclipse JDT for more than a year, except for P5 and P6, both of whom reported to be very comfortable with the environment. Seven of the ten had been involved in developing Eclipse since its inception (approximately four years). Eclipse is developed by a set of groups corresponding to the different subsystems within Eclipse; of the eight current Eclipse developers (P1-P8), only one had moved to another group. Each developer had their own fully-enclosed office with a door.

It is important to note that although all participants developed Eclipse, none were from the teams responsible for developing the Eclipse JDT. One developer was part of the 'Platform/User Interface' team, but this group defined the base level behaviours, equivalent to constructing advanced widget systems.

Participants were aware of the purpose of the study, though we avoided providing a description of what we meant by disorientation. One of the IBM developers, for whom English was a second language, required more detail during his final interview and was provided a description "it's when you feel lost: you can't remember what you were doing or what you were trying to find or why." We address the issues arising in Section 8.

3.2 General Format

Each participant (P1-P8) first completed a questionnaire about their software development experience. Participants were then observed for two hours as they pursued software development tasks of their choice. Participants were asked to think aloud [34], verbalizing what they were doing. Two interviews were conducted at mid-point and at the conclusion of the session.

The two non-Eclipse developers (E1,E2) were only interviewed; there was no observation.

Observation took place using both videotaping and screen-capture. Notes were taken of non-computer related actions and events, as well as any unusual comments made by the developer. The researcher sat behind and to the side of the participants, out of their line of sight, such that it was possible to observe what each participant did and to read documents on-screen to some extent. The video camera was placed approximately one half to a full metre away, aligned to capture the profile of the developer as they faced their computer monitor, with view of the keyboard and mouse. Developers were asked to think aloud, instructed to explain what they were doing as if to to another experienced developer. We hoped this would result in asking questions similar to *'now how did this fit in... ?'* signalling possible disorientation.

A brief interview was conducted after one hour to ask about the developer's progress and current approach. A longer interview took place at the end of the session to obtain more detail about their tasks, and how they found and kept track of the information relevant to their task. General questions and guidelines were prepared in advance of the interviews, and are described in the following section. All interviews were taped and subsequently transcribed word for word, including such features as pauses and inflection.

3.3 Questioning Convention

Participants were asked about their program navigation approach and and difficulties they encountered. Questions asked included: What information did you access? How did you find it? How did you know it was relevant? What tools did you use? Were any your own custom tools? Did they provide too much information, too little? Did you run into any problems finding the necessary information? What kinds of information do you wish could have been available?

Questions were also asked about the specifics of the change task. Only towards the conclusion of the final interview were the participants asked outright if they had experienced any episodes of lostness during the session: Did you become lost during the session or in previous sessions? What did that entail?

3.4 Method of Qualitative Analysis

Our analysis took several forms: analysis of the interview transcripts and field notes, and coding of the video sessions. The interview transcripts were first perused to identifying interesting comments. They were then read systematically, and with events and supporting quotes extracted and grouped into common themes.

The screen captures were analyzed to gather information on how developers use features of their development tools or other applications. This involved counting the number of transitions between Eclipse windows and other applications such as e-mail readers or web browsers, as well as the number of times where there was a total replacement of content — where information in a window became completely obscured.

The verbal protocols were used to confirm observations from the screen captures and notes.

3.5 Description of Developers' Tools

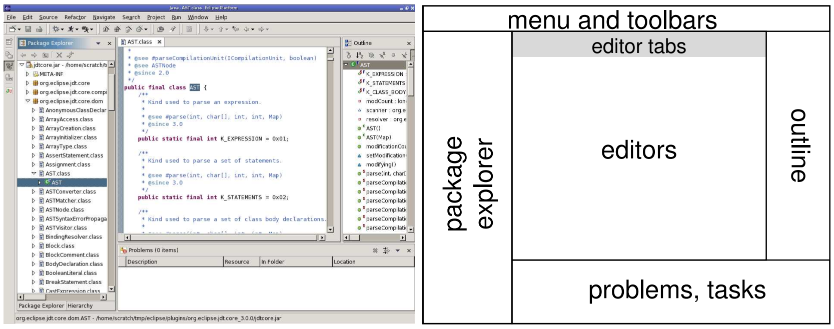
This section provides a characterization of the software development work observed during the study, including the tools used and some of the ways in which the tools supported the work. The Eclipse JDT is one of the primary tools used for development, and is very configurable by the end-user, and we provide a high-level description of the Eclipse JDT in Section 3.5.1. We then describe the other tools used during our study in Section 3.5.2.

3.5.1 Overview of the Eclipse JDT

The Eclipse JDT (herein referred to simply as 'Eclipse') provides state-of-the-practice features for searching and navigating through code, as well as for tracing through the dynamic execution of the program. We provide a very brief description of some key features of the environment, referring the interested reader to the Eclipse documentation [10].

Eclipse is *editor*-oriented, where each file is edited in a separate editor. The environment supports having multiple windows, each with their own independent set of editors. A window may also have a set of *views*, which are specialized GUI widgets providing information either about the currently selected editor or other state. The locations and sizes of the views and editors may be tailored by the developer/user, and a set of views and the editor location can be configured and recorded as a *perspective*.

Eclipse provides three pre-configured perspectives tailored to Java: the Java perspective (Figure 1), the Java Browsing perspective, and the Java Debugging perspective. The Java perspective, used by most developers for their Java development, features a central tree view named the *Package Explorer*. The top level of the Package Explorer tree corresponds to the different projects, then the top-level source directories, followed by the Java packages, then the Java class files and finally the actual classes. Although this organization requires a large amount of scrolling when dealing



```
 *
 * @see #parseCompilationUnit(ICompilationUnit, boolean)
 * @see ASTNode
 * @since 2.0
 */
public final class AST {
    /**
     * Kind used to parse an expression.
     *
     * @see #parse(int, char[], int, int, Map)
     * @since 3.0
     */
    public static final int K_EXPRESSION = 0x01;

    /**
     * Kind used to parse a set of statements.
     *
     * @see #parse(int, char[], int, int, Map)
     * @since 3.0
     */
    public static final int K_STATEMENTS = 0x02;

    /**
     * Kind used to parse a set of class body declarations.
     *
```

Figure 1: The Eclipse Java perspective as shipped with Eclipse 3.0 M7.

with classes in different packages, branches can be kept open showing more of the surrounding context.

Eclipse supports many sophisticated program navigation traversals such as finding all callers of a method, all references to a field, or the declaration of a class. Many such functions are tied to hot-keys, meaning they can be invoked with little effort. The environment also maintains a web browser-like stack-based traversal path, supporting switches between editors as to the order in which they were traversed.

Eclipse features an incremental Java compiler that rebuilds only the necessary parts of the program upon each file save. The compilation results are dynamically fed into views such as the *problems* view, which shows syntax errors across the source code; side-effects of changes thus become immediately apparent to the developer.

3.5.2 Developer Tool Use

The developers observed chose relatively small to medium tasks. Table 1 presents a rough summary of their work tasks along with a self-assessed estimation of the task complexity in comparison to their normal tasks; an asterisk (*) indicates that the task proved to be more complex than expected due to difficulties that arose during the task.

The developers used four other standard tools in addition to Eclipse: Lotus Notes for their e-mail; Lotus Sametime as an instant messenging service; Bugzilla, an issue-tracking system, for managing problem reports (PRs) and feature enhancements for Eclipse projects; and Google for searching documentation.

Developers also spent time on other related tasks during the study period, such as responding to queries from users and other developers via e-mail, instant messaging, visits by other developers, and triaging and responding to PRs. These other tasks were often interleaved during down-time, such as occurred during builds, waiting for Eclipse to start, or when fetching and committing files to or from the version control system. Although not formally verified, many of the e-mails received seemed to pertain to Bugzilla-generated e-mails in response to changes to PRs.

The Eclipse source code is managed by CVS [3] using facilities provided as part of the Eclipse environment.

	Task(s)	Complexity
P1	Ensuring backwards compatibility of reworked subsystem; fixing test-suites	Average*
P2	Communicating information between separated layers	Less
P3	Implementing last-minute minor features, bug fixing, constructing test suites	Less
P4	Porting task; modifying component to use lazy computation	Less
P5	Finessing Eclipse start-up appearance	Less
P6	Attempting to understand the interactions between a set of event listeners as part of larger task	Hard
P7	Porting task	Average
P8	Script verification; bug identification and work-around	Less*
E1	Business process product development	–
E2	Business process product development	–

Table 1: Summary of tasks observed with reported complexity as compared with other usual tasks.

4 Study Findings

During our study, we witnessed two episodes of disorientation, and two other developers reported experiencing disorientation (which was not apparent from our observation). We were surprised to not witness rampant disorientation. This made sense in hindsight: developers cannot afford to become lost too often. If the disorientation resulting from use of a tool was unbearable, developers would have either (i) abandoned the tools, or (ii) developed work-arounds.

To frame the discussion of the observations, we first discuss the manner in which we identified disorientation in Section 4.1, followed by a detailing of the occurrences of disorientation witnessed or reported in Section 4.2. We analyze these occurrences and provide an initial characterization of the types of disorientation in Section 4.3. In Sections 5 and 6, we examine different factors of the developers' environments and situations that we believe helped contribute to and/or prevent and/disorientation; we examine the contributing factors first as they provide grounding to the strategies. We suggest recommendations for tool support in Section 7.

4.1 Identifying Disorientation

One of our difficulties was determining how to characterize disorientation: how can we operationalize disorientation? It is difficult to identify disorientation simply from analyzing the study artefacts: furrowed brows, drumming fingers, and sighs, which occurred during the sessions, are also symptoms of problem solving. More importantly identifying disorientation requires some agreement as to what constitutes disorientation. Definitions varied across the developers: P4 believed that disorientation occurred when the UI did not behave as expected, such as window redraw delays, and P7 considered unexpected API behaviour disorienting. Although these unexpected behaviours may be disconcerting, they are better characterized as loss of situation awareness [12], as they relate to comprehending changes in the external world. Disorientation instead results from difficulties in the mental world, a difficulty in situating one's self in context.

We noted how developers tracked their locations, frustration expressed, their pace of work, and how developers switched between views. The verbal protocol provided additional clues, as users would sometimes ask themselves a question followed by a pause and perhaps a blank stare. When reported disorientation during the interviews, we asked for details so as to eliminate situations relating to situation awareness.

4.2 Disorientation Occasions

Several developers experienced disorientation during the study, summarized in Table 2. The row labelled '**Disoriented in study?**' refers to whether disorientation was observed during this study: **rep** indicates that disorientation was self reported during the interview, and **obs** indicates that evidence of disorientation was observed by the researcher. The following row, labelled '**Disoriented prev.?**', refers to a developer self-reporting having experienced disorientation in their past. The remainder of Table 2 provide details on the developers usage of Eclipse, which will be referred to throughout Section 5.

4.2.1 Disorientation occurring during the study

Four developers either reported or were observed to experience feeling lost during the study. After a prolonged period, P6 brought up a dialog to open a class (Open Type) and then momentarily blanked, unable to remember the name. We suspect he was mentally retracing his past steps in an attempt to recall the class name from the situation. P4 structured his development work using two different but similar-looking windows, one for making code changes and the other for modifying test suites. When returning to the code, having performed some testing, he was often visibly confused as to which window he should be using. P4 also said he must remember to later optimize some quickly hacked code, but did not write this down and was not subsequently observed to have performed the task.

P2 and P8 reported becoming lost during their development task. These occasions were not readily differentiable from their other actions. P2, who was working on a problem that involved several well-separated components, reported omitting one of these components accidentally more than once, and forgetting which operations were processed by which components:

> **P2:** When I was starting, the first time, I actually skipped a level. . . I do this frequently because we inserted a new level [. . .] months ago, but I still do it. I flip up to the runtime level and I should be at the [other] level. And then dealing with the command line, I always forget which things get processed at which level.

P8 reported getting lost in the file system and resorted to using very slow and considered movements to prevent becoming lost in some other directory.

4.2.2 Disorientation reported from previous occasions

Other developers (P3,P6,P8,E1,E2) reported having experienced previous occasions of lostness:

	P2	**P8**	**P6**	**P4**	**P1**	**P3**	**P5**	**P7**	**E1**	**E2**
Disoriented in study?	rep	rep	obs	obs	–	–	–	–	N/A	N/A
Disoriented previously?	yes	yes	yes	yes	yes	yes	no	no	yes	yes
# Files modified	5	8	25	6	1	13	2	2	–	–
Modifying own code?	75%	50%	no	yes	yes	yes	yes	yes	shared	shared
Thrashing	high	high	high	med	low	med	med	med	–	–
Customizes Eclipse	yes	yes	no	no	no	yes	no	no	yes	no
Runs Eclipse full-screen	no	yes	yes	no	yes	no	yes	no	no	–
Feels supported by Eclipse?	no	yes	yes	no	yes	yes	yes	yes	yes	yes

Table 2: General summary of observations and responses; explanations are provided in the text.

E2: Well sure, I can remember times sitting in front of the screen thinking, what am I looking at?

P3: When I'm [. . .] in browsing mode, I'll totally forget how I got to certain places.

Several noted that they rarely became lost with code they were familiar with (P2,P3,E1):

E1: [When starting on this project] lot of the code was new to me so I had to browse a lot to see 'where was this class,' 'who references it,' 'what does it really do here.' Sometimes you go into [a] method [from] another method, and I have no idea where you were coming from any more, you forgot, you lost track. But that usually happens, from what I remember, with code that I wasn't very familiar with.

Only P5 and P7 could not recall any occasion of feeling lost.

4.3 Characterizing Disorientation

In examining these incidents, we believe that the disorientation exhibited by developers is instead better characterized as two related sub-problems: navigational and task disorientation.

Navigational disorientation. We characterize this as when the developer:

- has lost their sense of location and direction (*what am I looking at?*);
- is unable to access an item of interest (*why can't I find. . .*);
- has lost their past history, unable to remember how they came to their current or past locations (*what was I doing?*).

Task disorientation. We characterize this as when the developer:

- cannot remember the task intended once at a location (*what was I going to do?*);
- has pursued or was distracted by another, perhaps related, problem (an *embedded digression* [14]), but failed to return to the original task.

4.4 Comparison with Disorientation in Hypertext

In forming this characterization of disorientation in software development, we considered both navigation problems as well as the developer's intent (or task). Our resulting characterization is similar to that resulting from the efforts in examining disorientation in hypertext. However, we specifically exclude the information management problem category. In the hypertext categorization, this category generally recognizes a problem styled as "the art gallery problem" [14], where a browser encounters difficulty summarizing what has been browsed.

In eliminating this third category, we follow the reasoning of Mayes et al. [23], whom emphasize a difference between disorientation in a spatial network (the structure of the information space) and in a conceptual space (the underlying structure of the information space). Some conceptual disorientation should be expected during knowledge-discovery tasks, where one's current understanding must be continually updated with new information garnered about the information space. Such learning forms a large part of the software exploration process, as developers explore the source code to update their understanding of the sometimes-complex causal relations in the system code.

5 Factors Contributing to Disorientation

We examined the situations around the occurrences of disorientation in detail to identify a number of factors that contributed to disorientation. Some factors relate to displays with low visual momentum, which cause difficulties for developers to rebuild their context. Other factors relate to developer habits.

5.1 Lack of Navigation History and Context

All developers made heavy use of Eclipse's code navigation operations. A single key combination can reveal the declaration of a particular method or find all references to an item (a field, method, or type), providing for quick descent in a call chain. Four developers (P1,P3,P5,P6) were observed to use these features often during their exploration. But as the editors are replaced, and the views generally update to the new file, there is no visible indication of how the developer came to a file.

P6: Yes, I do find [the code paths traversed] difficult to remember. One of the things, Eclipse makes it so easy to go down the call path, that it's hard to keep track of it. Whereas if I'm jumping from method to method, you lose context of what class you're in, because all there is is a title at the top that's flipping. [. . .] It's not clear as to what path you're actually following. [. . .] There's a broader view to it that you don't get just by jumping methods.

P8: I think if the navigation was not easy, I would more carefully remember these details.

These "flipping titles" were a source of difficulty. Eclipse manages each file as a separate editor, and the most recently used editors are available as a set of tabs (e.g., Figure 1).

E2: I'm just not interested in [using the editor tabs] because it's always a mess. [. . .] I don't know what's going on in there in my mind, so I just don't use them.

Only a fixed number of tabs are able to be displayed at any one time and any remaining tabs are hidden, as determined by a least-recently-used basis. But the order of the visible tabs bears little relation to when they were used: the file list is maintained as a queue, where newly-opened files are added to the end of the list. The tab order corresponds to the file list order: Thus navigating through a call chain that traces into previously-opened files will not cause those tabs to open at the end, but in their previous position, and thus destroy any relation to other visible tabs.

Consider, for example, searching for all references to a method, an action often observed during the study: although the Search view shows the matches (e.g., Figure 3), the location from where the search was initiated — which provides the reason for the search — is rapidly "lost" once having examined more than a few of the matches.

Having finished examining the matches, the developer may no longer remember why he was examining them; P8 described an attempt to rebuild his mental context when lost:

> **P8:** But when I'm lost, what I tend to do is go Alt-Back, back a row. "I'm lost, give me my context back! Where [am I]?" And after three or four things, if I'm not coming back on what I was expecting, I close everything.

P8 was attempting to rebuild the context as to how and why he had come to a location—trying to recover the relationships that led him to a particular file, in the hopes that this would remind him of the task being pursued. Because these relationships were not somehow externalized, the developer must remember them or rebuild them. If this recovery failed, he would close everything and restart from a known place.

These problems explain the behaviour observed of P5 and P6, both of whom had been engaged in a long period of source exploration, whom periodically cleared their lists of files which they had not modified. These files were ones they had browsed but that were not deemed important. Thus their file lists contained only important files.

5.2 Peering Through a Keyhole: Thrashing to Obtain Necessary Context

Although Eclipse provides some support for examining inter-file relationships without replacing the currently-displayed editor, such as showing the Javadoc for a called method in a tool tip, there is generally little visual momentum apparent for understanding the inter-file relationships and the larger organizations of the system. Developers are effectively limited to a single editor, and thus examine the system as a whole through a limited *keyhole* view [32]. This leads to the total replacement and thrashing problems observed.

P2,P4,P6 expressed a common desire to maximize the available screen 'real-estate' for viewing information, evidenced by the number of developers who use Eclipse to occupy the full-screen or near full-screen (see Table 2). Although primarily referring to increasing available editing space, allowing more of the surrounding file to be viewed, this complaint also speaks to minimizing wasted space on user interface elements that serve no purpose.

> **P2:** My biggest challenge is that I only have one view of the world, and it's because [Eclipse shows] one editor. And I can't see the different pieces of code, and it drives me crazy.

> **P6:** I can't see enough information in files that I'm working on.

P4 was the only developer observed to use more than one Eclipse window, but the two windows were nearly full-screen and he was sometimes momentarily confused which window he was using. P3 and E1 used separate stripped-down perspectives when editing code, as compared to when understanding code, so as to make more of the source code available. P2 and P3 noted continually experimenting with changing their Eclipse window layouts to increase their available real estate.

P2,P4,P8 exhibited significant *window thrashing* [18] during their tasks, repeatedly scrolling and jumping to particular points both between and within a file.

> **P2:** So basically I'm scrolling all the time, between the top to set a constant, the middle to read the constant, and the bottom [for] processing related to the constant.

Thrashing occurs when the content from several overlapping windows for necessary to the task, but cannot be seen simultaneously. Switching between overlapping windows causes replacement of content, and even lead to the *total replacement* of previously visible content in the window, forcing developers to maintain more information in their working memory. We assigned a subjective rating of the thrashing exhibited by each participant (Table 2), with *low* indicating little to no thrashing, *med* for some thrashing, and *high* for wild thrashing.[2] The switching generally involved switching between Eclipse and either Bugzilla or their e-mail application to record information, or between files. The thrashing generally occurred when they appeared to forget information to be copied across, having to return to the original source.

Interestingly, none of the developers used Eclipse's Bookmarks feature, which allows assigning a bookmark to a particular line of code. A central bookmark list would then provide for simple navigation between the bookmarked code. When asked, P2 indicated that he had enough difficulty keeping his windows to a usable size, and was not interested in using yet another view.

5.3 Pursuit of Digressions

Developers are often managing several tasks at any particular moment. Some tasks are suspended to pursue subtasks, called an embedded digression [14]. These digressions are usually related to the task: for example, attending a problem report may require some exploration to view the implications of a change. Some digressions may become more substantial: we observed fixing or diagnosing bugs in integration builds (P1,P8) and test suites (P1), and cleaning up old code mid-stream (P6). A digression may itself spawn further digressions, such as might happen when unearthing a bug: P1 and P8 encountered significant problems in their work, and what they expected to be five minute jobs took more than an hour and a half.

Developers were observed to pursue embedded digressions without recording the task they had suspended. Because Eclipse had no knowledge of this digression, the developers had to remember to resume their suspended task, which is known to be unreliable [11, 19]. One typical means to manage such digressions is to use separate application windows to manage each task context [8]. This solution was not used by any of the developers observed, most of whom used one window exclusively. Pursuing embedded digressions results in a clash between the different task contexts as the the same environment is used to represent both tasks.

5.4 Lack of Code Familiarity

Developers reported becoming disoriented when looking at code with which they are not familiar (P2,P3,P6,P8,E1). P6 spent his session trying to understand the causal model of a new set of code:

P6: Because what you are really trying to construct in your head is essentially a class diagram: you're trying to see how these classes interact. You're trying to what contains this

---

[2] This subjective measure was assessed after counting the number of transitions between Eclipse windows, different files, and other applications, as well as the number of times where there was a total replacement of content. We did not incorporate a measure of time nor did we take into account the context to eliminate false positives. For example, some developers would process e-mail while waiting for a regression test pass to complete; although this would technically result in total or near-total replacement, it was not a genuine switch as the old information was not being carried forward.

class, and uses it for this particular function. And the names of the methods being called is kind of a secondary aspect.

Developers reported that developing models of unfamiliar code was difficult:

P2: It takes me a while to build the model of where everything is, but then I got that model, and I'm pretty good at having a pretty extensive model. It takes me a while to build it, and it disappears very quickly, but that part's OK.

E1: Lot of the code was new to me so I had to browse a lot to see 'where was this class,' 'who references it,' what does it really do here. Sometimes you go into method into another method, and I have no idea where you were coming from any more, you forgot, you lost track. [...] it was huge, I had no idea how the pieces worked together.

5.5 Little Use of External Records

As part of their work, developers must correlate a large amount of information about the relationships within the source code. Although none of the developers were observed to write down or reference any external information, with the exception of P8 who maintained a simple paper task list, we suspect recording information externally might provide a way to alleviate disorientation.

When asked about his disinclination to record information, P2 was baffled, and asked *"What would I write down?"* P5 explained that he had used his whiteboard and paper the previous week when trying to gain an understanding of some pieces of code. P6, who only managed to find his log book after some rummaging through a drawer, explained that he does a lot of writing *"when I'm first thinking of something"*, but that he knew what he was doing at this point; P2 commented similarly.

This reluctance seemed surprising, especially for those developers attempting to understand unfamiliar source code. E1 explained that he would sketch relationships on paper when first exploring code, but he does not typically refer to them afterwards, and was unable to find his last sketches. Developers instead use the tools in the environment and instrumenting their code to derive any necessary information. P6 and P7, whose tasks required understanding how some other code worked, relied on Java `System.out.println()` due to subtle timing issues in their code.

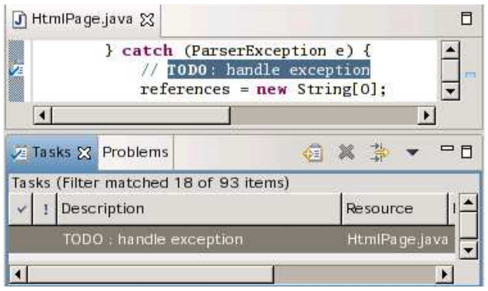
6 Strategies for Remaining Oriented

Our detailed examination also identified a number of strategies or work-arounds taken by developers to minimize disorientation.

6.1 Explicit Task Management

Eclipse development is issue-based, where each issue is generally tracked by a PR. PRs may be entered by the community or by other developers, which are then triaged and assigned to the developers responsible for the components. Many developers use Bugzilla as their to-do list (P1,P2,P6):

P1: I am typically someone who lives and dies by the problem report. I tend to put a lot of information into problem reports: I'll hang patches off of them, that sort of thing. I use Bugzilla as my sort-of to-do manager more than anything.



```
        } catch (ParserException e) {
            // TODO: handle exception
            references = new String[0];
```

Tasks (Filter matched 18 of 93 items)

✓	!	Description	Resource
		TODO : handle exception	HtmlPage.java

Figure 2: `TODO`s are captured by Eclipse and displayed in the Tasks view.

These PRs provided the context of their current work, as well as providing a place to capture additional information related to the task. The use of Bugzilla as described above by P1 may answer the question posed in Section 5.5.

6.2 Avoiding Digressions

Some developers avoided digressions as much as possible, instead noting items requiring further investigation. These digressions may be subtasks of a larger task, such as the PRs described in the previous section. For example, P3 and P4 used Eclipse's light-weight `TODO` markers (see Figure 2) to describe issues in-line with the code.

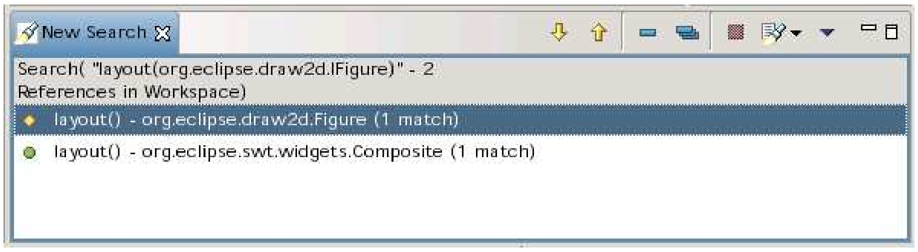
> **P3:** I thought of something I need to do, and I don't have to pickup a pen and piece of paper, I don't have to switch to another editor, I don't have to go to Bugzilla, just right in my code I say `TODO`, the keyword in the comment, write a blurb, go to the next line, continue on.

These would be picked up during the clean-up stage before committing their changes to the source repository. Other methods observed included using Bugzilla (P6 would use Bugzilla for even tiny tasks) or maintaining separate lists (P3,P8).

6.3 Using Environmental Cues to Serialize Work

We observed P3,P4,P6 use the Eclipse search facilities (Figure 3), which provides lexical and structural search functionality, for serializing block changes. P4 noted that many changes involved replacing a similar set of code, which could generally be identified by the results of a search query, such as to references to a particular call or access to a field. Having first pruned the query results of any extraneous elements, the developer would systematically modify each element and then remove it from the list. P4 would often make two passes, leaving difficult locations requiring thought for a second pass. The Eclipse search view provided a cue as to the ongoing task

Being systematic in this way helped the developers to stay on track: if problems were found during investigation, several developers (P3,P6,E1) would note it as a future task, filing a reminder in Bugzilla or as a `TODO`. P4 was particularly emphatic about not straying from his current task, noting that digressions often confound testing, as it complicated identifying the cause should a problem occur.



```
Search( "layout(org.eclipse.draw2d.IFigure)" - 2
References in Workspace)
layout() - org.eclipse.draw2d.Figure (1 match)
layout() - org.eclipse.swt.widgets.Composite (1 match)
```

Figure 3: The Search dialog was observed being used to serialize a set of modifications.

Griswold et al. [16] noted a similar unanticipated use of their AspectBrowser tool, where the developer used it to identify a set of relevant locations in the system code, which he then systematically worked through.

6.4 Avoiding Unfamiliar Code

Developers reported actively avoiding any need to understand new code. P5 was observed to look for examples rather than trying to understand the internals of the package itself. E1 and E2, both of whom used several external packages, commented that they both did the same. Developers avoid browsing code written by others to avoid having to learn new code, saving both time and effort.

This is reflected in the organization of the Eclipse development. The Eclipse developers (P1-P8) were part of groups where code ownership was a strong concept; a developer typically modifies only their code and not that of others. Developers are thus very familiar with their code, but less familiar with code of other groups or even their teammates. This generally was not a problem as they would simply ask the responsible developer for an internal component (P1,P3,P4,P8).

By restricting themselves to their own code, or code they have become familiar with, developers are less likely to suffer disorientation (reported by P2,P3,E1):

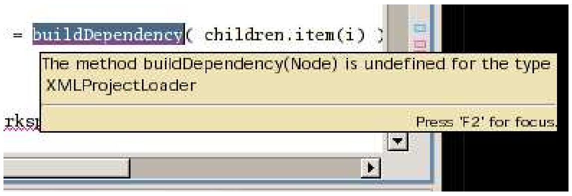
P3: I'm intimate with the classes, [. . .] I wrote these classes, I've been all over them and I know what's where and who's where, so I know how to navigate to get to them.

This behaviour is consistent with Mantei's finding that familiarity with the network aided in preventing disorientation [22].

6.5 Using Environmental Supports to Remain On Task

We observed developers using many of the non-obscuring information presentation techniques provided by Eclipse. These techniques, used for displaying information about inter- and intra-file relationships, remove the need to move away from the current position. Three examples of such features are:

Tool Tips: All developers were observed to use Eclipse's tool tips in some manner, which are used to display helpful information about the source code, such as error text or auto-completion suggestions (e.g., Figure 4). For functions that modify code, an additional tool tip window is used to show a preview of the



```
= buildDependency( children.item(i) )
```

The method buildDependency(Node) is undefined for the type XMLProjectLoader

Figure 4: Hover-style error description.

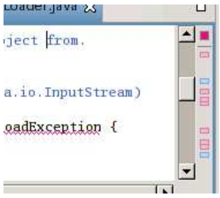


Figure 5: An editor outline bar featuring several markers (found to the right of the scrollbar).

change. These tool tips serve as functional overlays [32], providing supporting information without obscuring or replacing the code.

Editor Outline Bar: This bar provides a compressed overview of the file contents (Figure 5), showing the locations of relevant markers such as errors, warnings, `TODO`s, and breakpoints. With the appropriate option enabled, it will also show cross-references to a selected field within the file.

Problems View: This view presents a list of the outstanding errors, and is an example of Eclipse's dynamically-updated views. Most standard Eclipse views respond to changes as they occur, meaning that side-effects of changes are immediately available, supporting the principle of immediacy [31]. For example, saving a file triggers the incremental compilation, thus updating the problems view with any syntax errors found, and updates the tasks list with new `TODO`s (as used in Section 6.2).

6.6 Use of Spatial Dedication

Spatial dedication, being able to lay out information in a meaningful fashion, has proven to be a useful technique in other domains [18, 32], leveraging humans' automatic encoding of spatial and temporal attributes in memory [17].

P2 described using knowledge of the spatial location of certain resources during his task:

> **P2:** I was managing some of [the necessary information] spatially by knowing what order, not thinking about it, but inherently knowing where the editors were for different things. [. . .] I don't know the name of the thing, but I know where it is.

P5 *knew* that a certain method was always at the bottom of one file.

7 Recommendations

From our observations and interviews, we present a set of recommendations to developers and IDE designers to help alleviate disorientation.

7.1 Recommendations for IDE Designers

The following recommendations take a different focus than adding new views. Many of the problems with navigational disorientation arise because of difficulties in exploring inter-file relationships. We believe new specialized views are unlikely to cure the problem. Developers are extremely reluctant to use new views because they consume valuable screen real-estate, and also impose a learning curve. New views are rarely used in isolation, and may cause unintended effects when used in practice — even contributing to disorientation. Visual momentum provides a heuristic to assess the effects of new views as developers accomplish their tasks.

Support Examining Second-Order Relations

A primary component of development tasks requires correlating large amounts of information to gauge the impact of changes on other areas of the source code. Determining these impacts may require investigating not just the direct relationships, but several degrees of separation from the possible changes. We have already noted a number of Eclipse features that provide high visual momentum *within* files; for example, the outline bars provide an overview of useful information within a file, and the tool tip style windows present related information without obscuring the current content of interest. But these features are not always sufficient, especially for between-files relationships. Performing these types of investigation usually requires several several navigation steps, which generally causes the source of the search to be lost due to content replacement, and the actual relationship navigated to transition is usually lost as well. IDEs should provide support in determining and remembering these second-order relationships, and save the developer from carrying this mental burden.

Encode Explicit Support for Digressions

Some digressions are necessary. As described in Section 5.3, developers were observed to pursue embedded digressions without recording the task they had suspended. The exploration of second-order relationships described in the previous recommendation can be considered as a minor digression. The difficulty is that the tools are not aware that a new task is being pursued [8, 18]. IDEs should provide some support for developers to pursue and recover from previous digressions.

Eclipse's ability to clone a window is useful, but not sufficient to *freeze* a task: such a snapshot requires more than navigation history, but must also include development history. P1 was observed to make a complete copy of his workspace before pursuing a digression so as to keep a known last point. He could not commit his accumulated changes to CVS as the team had a policy that less than pristine source code could not be committed to the main branch. But he noted that CVS did not provide sufficient support for this type of lightweight checkpointing, as branches were felt to be too heavy weight (also noted by P4,P7).

Support Developers in Using Spatial Dedication

IDEs should support developers in being able to spatially arrange elements in their display so as to leverage their spatial reasoning in understanding relationships. P2 described how a previous development environment allowed for him to arranged his code in a manner in line with his thinking about the problem domain, mimicking the physical or structural layout:

P2: I really want to be able to use my spatial abilities and say [. . .] *this* is stuff from level one, *this* is stuff from level two, and put them there, and have those things mean that.

But both P2 and P4 commented at length on their inability to accomplish this rearrangement with Eclipse as they could not configure the UI to a small enough — yet useful — size to allow multiple non-overlapping windows. In fact, four of the eight Eclipse developers run with their windows at full-screen, two others at nearly full-screen, and the remainder at sizes too large to have two non-overlapping windows at their current sizes.

7.2 Recommendations for Developers

In addition to the strategies in Section 6, we make the following recommendations to developers:

Externalize Information: Systematic use of PRs and writing down more information helps off-load the memory burden. This also supports recovery of context upon returning from a digression or interruption.

Use Multiple Windows to Manage Digressions: One typical means to manage digressions is to use separate application windows to manage each task context [8], thus leaving the context represented by previous windows undisturbed.

8 Possible Threats to Validity

In this section we address limitations and possible threats to the validity of our results. We describe how we countered for possible threats from each of internal validity, external validity, as well as construct validity .

8.1 Construct Validity

Threats to construct validity refer to possible effects from the setup of the study. Such threats may include effects from prior knowledge of the study on the developers.

Participants were aware of general purpose of the study, which could have two possible effects. This knowledge may have heightened their sense of awareness—and prevented disorientation. But four of the eight developers reported or were observed to have had difficulties; our observations are at worst a minimum to be expected. This awareness may have conversely caused the developers to experience more disorientation than would have normally. But eight of the ten developers reported having become disoriented on a previous occasion. All questions about disorientation were left to the concluding interview so as to reduce any influence on the sessions.

To address the possibility of evaluation bias, none of the participants were developers of the Eclipse JDT itself. The experimenter was introduced as a former employee of the organization, to counter the possibility that developers would avoid providing criticism to an outsider. All developers provided extensive feedback on perceived problems, and compared and contrasted features of other development environments.

8.2 Internal Validity

Threats to the internal validity refer to whether our observations actually represent the real problem, or arise because of external influences.

Developers may have arrived with a very particular interpretation of 'disorientation' due to the prior effort to deal with disorientation issues in Eclipse (as mentioned in Section 1). To ensure no confusion about the disorientation described, the experimenter probed for detail, and eliminated from consideration any situations that were not due to disorientation.

There may have been some impact from asking developers to think aloud. To ensure the validity of the verbal protocol, the developers were asked to describe their actions as to another developer, such that the protocol was expressed in the concepts already used by the developers, as prescribed by Boren and Ramey [4]. P4 felt the session was artificial, that the think-aloud may have changed how he worked. There may have been some evidence for this as he initially proceeded to implement the wrong structure, perhaps due to being distracted from his intended task. As a result, P4 was not willing to check in code or proceed without submitting the changes to thorough inspection. He was able to complete his set tasks, however.

8.3 External Validity

External validity pertains to the extent to which our conclusions may be generalized to other organizations.

Although we did shadow multiple participants from different groups, our observations are limited to one field-site. Future work would include choosing different samples, including female developers.

Two developers (P4,P8) chose simple problems, to be sure they could be done within the two hour limit; this was only discovered during the final interview. Despite this, both developers encountered some difficulties during their investigation; P8 encountered great difficulty.

Finally, an obvious risk to this study is that our results may only apply to the Eclipse UI, and not for software development environments in general. We believe this is unlikely: the mentions of disorientation in the literature with environments other than Eclipse confirm that this occurs with other tools, such as the developer frustrations with Microsoft VisualStudio reported by DeLine et al. [9]. The Eclipse JDT is an award-winning development environment,[3] and the Eclipse platform serves as a base for development environments for several other languages (e.g., C and C++, COBOL, web tools).

9 Conclusions

This paper has examined the phenomenon of disorientation through observation and interviews with ten IBM software developers working on the Eclipse project. Although

---

[3] `www.eclipse.org/community/awards.html`

disorientation was a rare occurrence during our study sessions, we found sufficient evidence to characterize disorientation in software development, resulting in a clarification of the difference between navigational and task disorientation.

Based on observations from our study, we have postulated a set of hypotheses, which identify factors that we believe contribute to preserving orientation or inducing disorientation. These factors should be considered in software tools design. These hypotheses also provide a basis for further study. In particular, we believe obtaining more systematic measures of disorientation could help better characterize the forms of disorientation, which could lead to more specific tool recommendations.

Acknowledgments

Thanks are owed to our study participants: they were very accommodating and provided great insight into the problem of disorientation. We would also like to thank Elisa Baniassad for her comments on this paper.

References

[1] Belady LA, Lehman MM (1976). A model of large program development. *IBM Syst J* 15(3):255–252

[2] Bellamy RK, Carroll JM (1992). Restructuring the programmer's task. *Int J Man-Machine Studies* 37:503–527

[3] Berliner B (1990). CVS II: Parallelizing software development. *In Proc. USENIX Winter 1990 Technical Conference*, 341–352. Berkeley, CA: USENIX Association

[4] Boren MT, Ramey J (2000). Thinking aloud: Reconciling theory and practice. *IEEE Transactions on Professional Communication* 43(3):261–278

[5] Brooks FP (1995). No silver bullet — essence and accident in software engineering. *In The Mythical Man Month*, chap. 16, 179–203. Addison–Wesley. Originally published in IEEE Computer in 1987

[6] Conklin J (1987). Hypertext: An introduction and survey. *IEEE Computer* 20(9):17–41

[7] Creasy T (2001). Request for comments: Loss of context. Online document: `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/loss-of-context/`

[8] Cypher A (1986). The structure of users' activities. *In* Norman NA, Draper SW (Eds.), *User Centred System Design: New Perspectives on Human-Computer Interaction*, chap. 12, 243–263. Erlbaum

[9] DeLine R, Khella A, Czerwinski M, Robertson G (2005). Towards understanding programs through wear-based filtering. *In Proc. ACM Symposium on Software Visualization (SoftVis)*, 183–192

[10] des Rivières J, Wiegand J (2004). Eclipse: A platform for integrating development tools. *IBM Syst J* 43(2):371–383

[11] Dix A, Ramduny D, Wilkinson J (1998). Interaction in the large. *Interacting with Computers* 11:9–32

[12] Durso FT, Gronlund SD (1999). Situation awareness. *In* Durso FT, et al. (Eds.), *Handbook of Applied Cognition*, 283–314. Wiley

[13] Elm WC, Woods DD (1985). Getting lost: a case study in interface design. *In Proc. 29th Annual Meeting of the Human Factors Society*, 927–931. Human Factors Society

[14] Foss CL (1989). Tools for reading and browsing hypertext. *Information Processing and Management* 25(4):407–418

[15] Green TRG, Gilmore DJ, Winder R (1992). Towards a cognitive browser for OOPS. *Int J Human-Computer Interaction* 4(1):1–34

[16] Griswold WG, Yuan JJ, Kato Y (2001). Exploiting the map metaphor in a tool for software evolution. *In Proc. Int. Conf. on Softw. Eng. (ICSE)*, 265–274

[17] Hascher L, Zacks RT (1979). Automatic and effortful processes in memory. *J Exp Psychol: General* 108:356–388

[18] Henderson Jr DA, Card SK (1986). Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans Graphic* 5(3):211–241

[19] Herrmann D, Brubaker B, Yoder C, Sheets V, Tio A (1999). Devices that remind. *In* Durso FT, et al. (Eds.), *Handbook of Applied Cognition*, 377–407. Wiley

[20] Janzen D, De Volder K (2003). Navigating and querying code without getting lost. *In Proc. Conf. on Aspect-Oriented Software Development (AOSD)*, 178–187

[21] Kim H, Hirtle SC (1995). Spatial metaphors and disorientation in hypertext browsing. *Behaviour and Information Technology* 14:239–250

[22] Mantei MM (1982). *Disorientation Behavior in Person-Computer Interaction*. Ph.D. thesis, University of Southern California

[23] Mayes T, Kibby M, Anderson T (1990). Learning about learning from hypertext. *In* Jonassen DH, Mandl H (Eds.), *Designing hypermedia for learning*, 227–250. London, UK: Springer-Verlag

[24] Parunak HV (1989). Hypermedia topologies and user navigation. *In Proc. Hypertext '89*, 43–50. New York: ACM

[25] Singer J, Lethbridge TC, Vinson N, Anquetil N (1997). An examination of software engineering work practices. *In Proc. CASCON*, 209–223

[26] Soloway E, Lampert R, Letovsky S, Littman D, Pinto J (1988). Designing documentation to compensate for delocalized plans. *Commun ACM* 31(11):1259–1267

[27] Storey MA, Wong K, Fracchia FD, Müller H (1997). On integrating visualization techniques for effective software exploration. *In IEEE Symposium on Information Visualization (INFOVIZ '97)*, 38–45. Phoeniz, Arizona

[28] Storey MAD, Fracchia FD, Müller HA (1999). Cognitive design elements to support the construction of a mental model during software visualization. *J Systems & Software* 44(3):171–185

[29] Tilley SR (2000). The canonical activities of reverse engineering. *Annals of Software Engineering* 9(1–4):249–271

[30] Turetken O, Schuff D, Sharda R, Ow TT (2004). Supporting systems analysis and design through fisheye views. *Commun ACM* 47(9):72–77

[31] Ungar D, Lieberman H, Fry C (1997). Debugging and the experience of immediacy. *Commun ACM* 40(4):38–43

[32] Watts-Perotti J, Woods DD (1999). How experienced users avoid getting lost in large display networks. *Int J Human-Computer Interaction* 11(4):269–299

[33] Woods DD (1984). Visual momentum: A concept to improve the cognitive coupling of person and computer. *Int J Man-Machine Studies* 21:229–244

[34] Woods DD (1993). Process tracing methods for the study of cognition outside of the experimental psychology laboratory. *In* Klein, Orasanu, Calderwoods (Eds.), *Decision making in action: models and methods*, 228–251. Ablex

[35] Woods DD, Watts JC (1997). How not to have to navigate through too many displays. *In* Helander MG, Landauer TK, Prabhu PV (Eds.), *Handbook of Human-Computer Interaction*, chap. 26, 617–650. North-Holland, 2 ed.