NavTracks — Helping Developers Navigate Source Code

by

Robert Douglas Elves
B.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

NavTracks — Helping Developers Navigate Source Code

by

Robert Douglas Elves
B.Sc., University of Victoria, 2003

Supervisory Committee

Dr. Margaret-Anne Storey, (Department of Computer Science)
Supervisor

Dr. Hausi A. Müller, (Department of Computer Science)
Co-Supervisor

Dr. Daniela Damian, (Department of Computer Science)
Departmental Member

Dr. Janice Singer, (National Research Council, Canada)
Outside Member

Dr. Gail C. Murphy, (Department of Computer Science, University of British Columbia)
External Member

Supervisory Committee

Dr. Margaret-Anne Storey, (Department of Computer Science)
Supervisor

Dr. Hausi A. Müller, (Department of Computer Science)
Co-Supervisor

Dr. Daniela Damian, (Department of Computer Science)
Departmental Member

Dr. Janice Singer, (National Research Council, Canada)
Outside Member

Dr. Gail C. Murphy, (Department of Computer Science, University of British Columbia)
External Member

ABSTRACT

Software systems can be very complex, tasking the developer's ability to keep a consistent mental model. Software systems are often comprised of many lines of code and scattered across many different files, all located within a complex hierarchical file system. This hierarchy must be navigated when adding new functionality or refactoring existing designs. Due to this complexity, developers can easily become disoriented and lost within software systems. For this reason, better tools for source code navigation are required.

In this work we present NavTracks, a tool that supports navigating through software systems. NavTracks keeps track of the navigation history of software developers, forming associations between related files. These associations are then used as the basis for recommending potentially related files as a developer navigates the software system. We begin with a brief survey of current tools for navigation in electronic environments. This is followed by a discussion of the design, implementation and evaluation of NavTracks. We finish with proposed future work.

Table of Contents

List of Tables

List of Figures

Acknowledgement

A special thanks to Dr. Hausi Müller for recommending I consider graduate studies. It has been a great experience to work with some of the best researchers in my field of interest.

I am indebted to Dr. Margaret-Anne Storey for her patience and encouragement. Her guidance was instrumental in keeping me focused in a diverse field of research.

I would also like to acknowledge Dr. Janice Singer for her positive attitude and contribution throughout this research.

Finally, a warm thanks to my family for their support.

- Robert

Dedication

To Rachel

Chapter 1

Introduction

Software systems are complex and consist of many lines of code. Many systems are so complex that to understand the whole system in its entirety is an impossibility for a single person. Software engineers tasked with extending or otherwise modifying these systems chunk the system into manageable units of work such that the impact of their modification is understood. This style of work was coined Just In Time Comprehension by Singer *et al.* and observed in their study of software engineer work practices [1]. Singer *et al.* observed that when tasked with a system modification, software engineers often learn just enough about the problematic code and the potential impact of the proposed changes to complete the work successfully. The details of the code are often quickly forgotten and they must re-explore portions of the system as they revisit them [1].

The process of modifying legacy code has been called Software Evolution, Software Maintenance, and Software Refactoring. Each of these processes begins with a reconnaissance phase in which the software engineer tries to familiarize herself with the code affected by the modification. This phase is often called Exploration or Investigation and its purpose is to develop an understanding of what code needs to be modified to implement the requested feature. There are numerous theories describing how to gain an understanding of a system, but the end goal is to establish a clear mental model of the system. Inevitably the user must view and modify the source code. It is at the point of source code navigation that we recognize a need for increased efficiency.

When following the logic of a piece of code, this often requires visiting many different

locations within the code base. If the modification required has local impact, the related code may all reside within a single class held within a single file. More often, the work requires visiting many classes distributed among many files, requiring investigation of many potential branch points and nested method calls. The reason for this complexity can often be attributed to poor design, unanticipated features, or simply the limitation of the programming language itself. These three conspire to separate related code into different classes, files, and directories. Cross-cutting concerns [2], as they are called, are spread throughout the code base; first, requiring the developer to identify their location within the code, then subsequently navigating among them in order to make the necessary changes. Navigating this maze of method calls, variable assignments and conditional branch points has been compared to navigating the World Wide Web (Web) [3, 4].

When navigating the web, users can become structurally or conceptually disoriented among all the hyper-linked routes and suffer cognitive overload [5, 6]. Such problems are amplified when navigating source code as it is even more complex due to the large number of links between the various source code artifacts [7]. The cognitive load that results from this non-linear navigation activity can cause the developer to become distracted, disoriented and altogether lost [8].

1.1 Methodology

The present work examines current and experimental navigation tools followed by synthesis and application of this research towards a tool for efficient source code navigation. We first draw upon the literature to solidify our understanding of navigation in terms of what cognitive processes and tasks it entails. With a language of navigation defined, we move on to analyze tools for navigating electronic environments. The results of the literature review are then synthesized and a prototype built followed by evaluation.

1.2 Thesis Outline

Chapter 2 begins with a high level investigation into navigation as a task — what motivates our navigation decisions and strategies. This is followed by a discussion of what practical implications this has on designing for navigation of electronic environments. Chapter 2 finishes with a review of a number of tools designed to help users navigate within electronic environments.

In Chapter 3, the discussion in Chapter 2 is synthesized into design rationale that are then applied to the construction of a prototype tool for navigating source code. Chapter 4 details evaluation of this prototype.

Chapters 5 and 6 complete the thesis with a discussion on potential future work and general conclusions.

Chapter 2

Background

Navigation is a difficult concept to discuss due its many differing definitions found in the literature. We begin by identifying navigation and defining some terminology so that we can discuss navigation design clearly.

2.1 Information Seeking and Navigation

When we say "navigation" what do we mean? A number of activities have been identified as navigation but there is little consistency in terminology. Terms such as browsing, searching, scanning, seeking, and wayfinding have all at one time or another been used to describe navigation. Depending on the community of interest (e.g., Human-Computer Interaction, Information Retrieval, Architecture), the definitions of these terms can vary dramatically. Adding to this confusion are the similarities and differences between the physical and electronic environments.

The type of action required may differ, but the cognitive processes that drive our actions appear to be the same. For example, we 'browse' the clothing store to locate desirable clothing just as we may 'browse' the World Wide Web for desirable information. Although the type of action required differs, both activities appear to follow the information seeking task cycle as described by Marchionini [9]. In both situations there is some need or desire that motivates us - a goal, and a process of activity that is constantly monitored to gauge our progress [9].

With this observation in mind, we draw on both information seeking and physical navigation literature to help construct an understanding of *navigation*. In particular we rely on Marchionini's work, "Information Seeking in Electronic Environments" [9] for analysis of the motivation and strategies that influence our navigation decisions. We also refer extensively to Jul's work, "From Brains to Branch Points: Cognitive Constraints in Navigational Design" [10] for insight into navigation design founded in empirical psychological research.

2.1.1 Tasks Estimation

The tasks or goals we define are subject to a cost analysis before we take action [9]. Whether or not we proceed is determined by our estimation of the task's physical and cognitive cost. These estimates are based on perceived task complexity and the nature of our goals. Marchionini defines three continua upon which goals may be projected: specificity, volume, and timeliness (defined in the following sections). Where goals lie on these continua, along with complexity, establishes total task cost [9]. This in turn affects our decision of whether or not to execute the task, and if so, the manner in which to proceed.

2.1.1.1 Specificity

Specificity refers to the certainty of the goal. From the user's perspective, the specificity determines how hard it will be to know the goal has been reached [9]. If the goal is vague, it may be hard to determine if the goal is satisfied. If the goal is well defined, determining success or failure is less difficult. For example, if a developer is familiar with a code base and seeks a specific file, this is a well defined goal. If a developer is unsure of which files to modify, seeking these files would represent a task of lower specificity.

The specificity of the developer's goal will influence which strategy and subsequent tool is chosen for the task. Marchionini identifies two classes of strategy for information seeking: analytical strategies and browsing strategies. Analytical strategies are determin-

istic and formal, relying on the user's ability to recall properties of the target in order to form queries. Analytical search is often used to describe the query based seeking found in electronic environments. An example would be querying the development environment for all files that implement a specific interface. The user is forced to recall the exact name of that interface in order to enter it as a query. If properties such as the name are well known, analytical search may be the most efficient strategy. Browsing strategies are more heuristic and interactive in nature, requiring more of our attention and recognition faculties.

In Jul's work on navigation design, a correlation between specificity of the goal and strategy chosen can be seen. Jul identifies three activities (strategies) often associated with navigation: finding, searching, and browsing. The first two of these are differentiated by the level of certainty the user has in the whereabouts of the target destination (the specificity of the goal) [10]:

finding going to a location or object within the environment with confidence in the knowledge of where it is or how to get to it;

searching seeking a location or object within the environment that meets certain criteria without certitude in prior knowledge of where it is or how to get to it.

Jul's definition of finding parallels Marchionini's definition of *directed browsing* referenced extensively in the literature to describe navigation behaviour. Marchionini describes directed browsing as *systematic, focused seeking for a known item.*

Search, as defined by Jul, seems to encompass both Marchionini's *semi-directed browsing* and *analytical search* strategies. Marchionini describes *semi-directed browsing* as characterized by a "less definite target" and proceeds in a more ad-hoc/opportunistic fashion [9]. Marchionini's *analytical search* essentially describes the query based seeking often found in electronic environments. Jul too explains that under her definition, *searching* may not require spatial navigation but rather query-based mechanisms.

In terms of low levels of specificity, Jul uses the term *browsing* to describe the process of becoming familiar with the structure and organization of the environment [10]. In this

form of navigation there may not be a defined destination. This is the process of spatial knowledge preservation and is required for particular navigation tasks [10]. Spatial knowledge preservation is the act of "gathering environmental information concerning possible movement" [10].

2.1.1.2 Volume

If the current task is expected to require a large volume of data, this will cost the user in terms of the cognitive overhead required to process any additional input and judge its relevance to their goals [9]. Depending on the design of the navigation aid, the number of branch regions can increase dramatically as more data is presented.

A branch region is a point at which a decision can be made to follow alternate paths. In a file hierarchy, a folder is a branch region that presents many options (files). Each additional branch region requires the user's attention and evaluation. This makes the navigation process more complex and can lead to the user becoming disoriented [11].

By reducing the number of branch regions we can reduce the amount of cognitive overhead. However, users will not necessarily pick the navigation tool requiring the least amount of physical actions but rather one that requires the least cognitive computation time [10].

2.1.1.3 Timeliness

The last of the continua is timeliness, which Marchionini defines as the expected time to completion — how fast the tool produces the desired result. The scale is from immediate to longer time frames such as minutes. Immediate timeliness is usually characterized by high specificity and low volume — very accurate with little noise (extraneous data). These types of targets are often temporarily forgotten and are needed as an intermediate step in some larger task.

Jul's *finding* can be associated with a timeliness factor approaching immediate. Since the desired destination is known, expectations for immediate transportation are high.

2.1.2 Tool Selection

Where a tool lies on these continua determines its usage cost. If the cost outweighs the user's expectations, the tool will not be used. Personal information infrastructure (and the tools available) affect user's judgement when estimating the cost of seeking a resource [9].

Cognitive load is reduced when the tool is organized to suit the user's task. When the organizational structure of the navigation aid doesn't align with the present task, this has been known to cause wayfinding or navigational error [10]. Users will go out of their way (often incurring additional navigation) to organize their environment into a more task-structured form [10, 12]. For example, when tasked with packing envelopes, rather than pick each item from a random mix of items it is much more efficient to make piles of each different item type and place the piles in order of insertion in the envelope. With this organization one can easily keep track of which content has been placed in the envelope. In addition it also helps us judge what content has not been included. These organizational tasks can now be off loaded onto the environment rather than having to be consciously aware of them at every step of the process. This increases efficiency.

An example of the need for different organizational structure can be found in the directory hierarchy found on most computers and development environments. These structures are organized into folders labeled with names that were conceptually relevant at some point in time. In the case of software development, this can be effective for locating software related to specific functionality. However, often times the organization that has been imposed does not match the mental model of all developers. Inevitably there are alternative, perhaps more meaningful containment relationships, that could serve as the basis for an organization scheme. Depending on the task, this organization scheme may be entirely different.

In an effort to automatically align navigation tools with the task/intention of the user, many have begun to put the computer to work in providing dynamic navigation aids. The computational nature of the computer gives us the ability to process data and make changes to structure and content in real time. Using this capability, data is gathered and recommendations are made in order to help the user navigate. A great deal of research is being

done on helping people navigate the Web using this technique. In Section 2.2 we examine a few of these web based approaches and then look at similar solutions that have been built specifically for navigating software systems in sections 2.3 and 2.4.

2.2 Navigating the Desktop and the Web

2.2.1 UMEA — Kaptelinin

The problem faced on the modern computer desktop is that one must locate and coordinate many different types of information objects [13]. Generally to solve this problem we put commonly accessed and related resources together in a single folder. This has the disadvantage that some resources participate in many different projects. Secondly, files are not the only information resource that are referenced during a project. Web sites, emails, etc, are all required to complete a given task. Unfortunately, these resources all require the user to access them through different resource specific organizational structures.

To counteract this complexity a tool called UMEA (User-Monitoring Environment for Activities), developed by Kaptelinin, monitors the user's activity and organizes the resources used into project related pools. These pools of related resources can then be accessed in a *single location*. The pools are explicitly created by the end users. Each time a user switches to another task, he/she must switch on UMEA otherwise the subsequent events will be logged under the wrong task.

Of the 8 subjects which participated in an empirical evaluation of UMEA, only one individual reported "feeling uncomfortable about dividing his activities into separate projects" [13]. There were two main problems reported with this tool. First, was the need to manually clean up the interaction histories if the wrong task was selected. Second, some users had difficulty understanding the user interface — no additional information regarding this difficulty was provided.

2.2.2 Information Management Assistants — Budzik

Along the same lines as UMEA is Budzik's Information Management Assistants. By monitoring the user's activity, implicit queries are sent to information sources to retrieve information potentially related to the current task context [14]. Task knowledge is based on examining the text of the user's current application.

Evaluation of this tool consisted of ten researchers loading the last paper they wrote. Budzik's tool then provided each researcher with a list of recommended resources from the web based on observation of their paper. The researchers were then asked to judge the relevance of each recommendation. Eight out of ten researchers said that at least one recommendation would have been useful to them [14]. Four of the researchers indicated that resources that were previously unknown to them were recommended and would be used in future work [14].

2.2.3 TaskTracer — Dragunov

Dragunov's TaskTracer focuses on helping users recover from interruption and is based on the premise that, "the user's task context usually includes not only the current document, but also other documents [...] which were used in the past" [15]. TaskTracker builds task context through recording the various events as they happen in common productivity applications (email, wordprocessing, etc). Users must specify what task they are undertaking. As data increases, the hope is that TaskTracer will be able to accurately predict the user's current task. TaskTracer can restore the applications to the last state they were in when the task was suspended. This is in hope to ease the burden of returning to a task after interruption [15]. Unfortunately this research is in its early stages and the algorithm used for task detection/segregation was not discussed.

2.2.4 Path Model — Chalmers

As alluded to earlier, navigating the web and navigating source code are similar in many respects. For this reason, the work of Chalmers on his Path Model based recommendation system are of interest to the present work [16]. The Path Model is simply a record of all navigation events that take place. This path implicitly represents our particular interests. In its intended context of the web, two users could be identified as having similar interests though measures of similarity between their respective paths. Recommendations could then be made to each other based on these measures of similarity and where they currently are and have been [16]. This technique is similar to Collaborative Filtering [17] but differs in the fact that the Path Model is contextually relevant [16]. Recommendations are based on where the user currently is and where they (and others) have been in the past. Collaborative filtering simply looks at the resource and just presents recommendations based on input from multiple users without taking into consideration the users recent task activity/context.

Recommendations based on the Path Model start with recording the url of each site visited. Regular expressions are used to filter out unwanted urls. A sliding window of size ten is run over the history. The ten most similar segments to the current segment (based on similarity of urls) are chosen. From these, the urls are tallied, and ranked. Events that have happened recently are removed and the results are displayed.

2.3 Navigating within Eclipse

We have examined selected research projects focusing on helping users navigate their desktop computer or the web. In this section we investigate how *finding* is supported within the Eclipse environment. We concentrate on answering the following questions. What locomotional structures exist to help users gain access to known locations/information? What is their organizing principle? We do not address methods based on Analytical Search although they are viable navigation tools. The present work focuses on tools with spatial locomotional structure that support finding and directed-browsing.

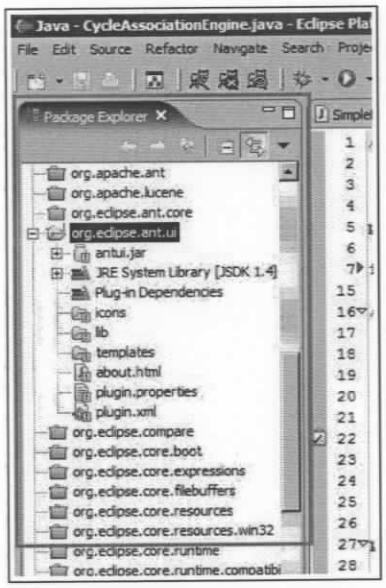

Figure 2.1. *Eclipse Package Explorer*

2.3.1 Package Explorer

Development environments such as Eclipse often facilitate movement between source files by providing a hierarchical tree based locomotional structure. This structure closely resembles the model used by the underlying file system (See Figure 2.1).

This thin veneer over the file system is, for many, easier to use than the command line when locating and loading files. The need to *recall* a file's location from a mental map of the directory structure is negligible since the interface is itself a very literal yet visual translation of the directory structure.

As the software system grows in complexity, the number of files and folders displayed in the Package Explorer can quickly become unmanageable. Each additional folder introduces a new option which requires evaluation and possibly the evaluation of each item contained therein. As the number of branch points increases, navigation efficiency is reduced[10].

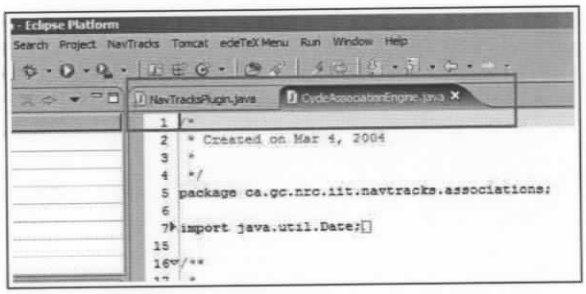

Figure 2.2. *Tabs in Eclipse*

2.3.2 Tabs

One example of a simple locomotional tool found in Eclipse and many other IDEs are Tabs. Eclipse Tabs, as depicted in Figure 2.2, offer quick access to open files. Files that are loaded (often using the package explorer) are opened and displayed with the tab either above or below the document depending on the configuration. This tab offers one click access to the associated file and can also serve as a reminder of what files the user is currently working on. The importance of the finding and reminding functions is revealed in the work of Barreau and Nardi [18]. As each file is opened a tab is added to the tab pane.

A few problems arise when the number of tabs exceeds the space available. First, the file name labels on each tab get truncated allowing more tabs to be displayed. This truncation of labels makes each tab difficult to identify. Second, as even more files are opened (and automatic close isn't enabled) excess tabs that haven't been viewed recently are removed completely from the tab pane. These excess file tabs can be accessed via a quick view (CTRL-e) that lists all loaded files with the recently removed file names highlighted at the top of the list as depicted in Figure 2.3. When hidden in this manner, the tab is no longer easy to access, nor can it function as a reminding aid.

A tab's global position within the display is not constant. As new files are opened and stale ones moved to the quick view, the tabs automatically shift from right to left. For this reason tabs must constantly be re-located, potentially disrupting the user's work flow. If the user manually positions two tabs together (perhaps by dragging and dropping one

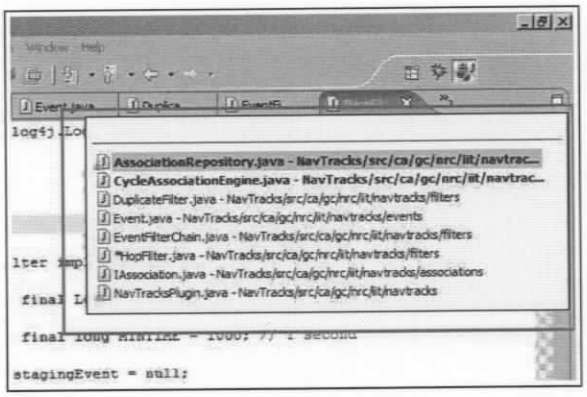

Figure 2.3. *Quick View*

tab alongside another), their relative position together will be maintained, but their global position within the tabbed pane can change if one or more tabs are closed to the left of the related tabs.

2.3.3 History

The Navigation History feature in Eclipse allows the user to step backward in time to previously visited files. The interface is a simple list organized chronologically and duplicate entries are not removed. The list is dynamic but changes in a very predictable manner.

2.3.4 Outline View

The Outline view, as depicted in Figure 2.4, offers a hierarchical view of the currently active source code file. The hierarchy presents fields, methods, and internal classes that are contained within the source files. This view is organized in order of appearance within the source file with the option to sort alphabetically. Selection of a program element from the hierarchy directs the editor to the location of the chosen element within the source file.

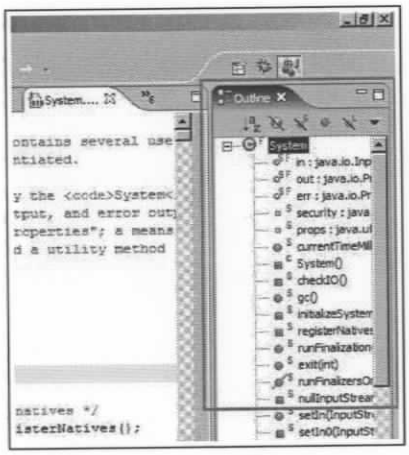

Figure 2.4. *Outline View*

2.3.5 Cross Referencing

Finally, it is possible to move throughout the source code by following method calls in a similar manner to following hyperlinks on the web. Holding down the CTRL key and clicking on a field or method will bring the user to the definition of that resource be it in the same file or another file. Resources are opened to reveal the target file if not already opened.

2.4 Research Navigation Tools

Due to being open source and its extensible architecture, Eclipse can be easily extended by the development community. A number of researchers have chosen Eclipse as a platform for testing new ideas in source code navigation. We will now look at tools developed by researchers to help developers navigate source code, many of which are built upon the Eclipse platform.

2.4.1 Change Patterns - Ying

One of the most common research topics in source code navigation is recommendation and collaborative filtering. These approaches either mine historical data from source code repositories such as CVS (Concurrent Versions System) or combine the information from these sources with recommendations from other users. This makes a great deal of sense since CVS is a tool which is common to many development efforts and tracks a great deal of historical data. These tools work on the intuitive assumption that files which are co-edited are often related. The more frequent two files are modified together the stronger the relationship. Ying [19] and Zimmerman [20] are examples of independent efforts which mine CVS for frequent co-commits and make recommendations. Both efforts have shown the potential of this method.

Ying's method consists of:

Identifying atomic commits Ying's first step is to determine what files make up an atomic commit. Ying defined an atomic change set as files that were committed with the same check-in comment and within three minutes of each other [19].

Filtering The second pre-processing step involves filtering the noise. Examples include atomic change sets that consisted of too many files to be of value (Ying chose 100 as the threshold).

Association Rule Mining The next step in Ying's process is to mine association rules from the change sets identified in the previous steps. The goal of this process is simply to identify sets of frequently co-committed files. There are many methods to solve this problem and Ying investigates two methods, one based on frequent set mining, the other on correlated set mining.

When looking at frequent set mining, Ying tried two different algorithms: the Apriori algorithm and a Frequent Pattern (FP) tree mining algorithm.

Apriori was discarded due to the potential inefficiency which results from its breadth first approach. The Apriori algorithm applies a breadth first approach in order to

find all sets of the same frequency before finding larger sets [19]. For this reason Apriori was discarded in favor of a FP-Tree based algorithm that performs a depth-first, divide and conquer approach to mining frequent patterns.

The second method employed by Ying is correlated set mining. This approach seeks to identify all sets of files with sufficient correlation, where correlation is measured based on a chi-squared test [19].

Query In order to produce recommendations, the developer must in some manner provide a starting file from which to base these recommendations. During the query phase, recommendations are gathered by finding sets mined in the previous stages that have the file of interest [19].

Evaluation was based on the predictability and interestingness of the recommendations returned by the system. Source code modification tasks were documented for two large software systems and compared against the modification recommendations made by Ying's system. Historical project data was broken into training and test data. Each test was done by providing a single file as a seed. Multiple files could be submitted but one file is the least amount of information necessary to get results [19]. Based on the training, actual file modifications were compared with recommendations in order to determine their level of predictability and interestingness.

Predictability is measured in terms of precision and recall. These measures are often used to measure the suitability of information retrieval algorithms. Ying explains:

> The precision of a recommendation [...] refers to the accuracy of the recommendations and is measured by the fraction of recommendations [...] that did contribute to the files in the solution [...] of the modification task,[...] The recall of a recommendation [...] refers to the coverage of the recommendations and is measured by the fraction of files in the solution [...] that are recommended...

For *interestingness* Ying uses a qualitative scale — surprising, neutral, and obvious — to judge the *interestingness* of a particular recommendation. These values are determined

based on structural and non-structural information gleaned by the developer.

Structural recommendation levels are fine and course (method, class, file). Using this methodology, Ying was able to achieve approximately 50% precision and 20-30% recall [19]. This means that 50% of the recommendations made actually matched files that participated in the solution. The precision of lower levels of granularity (class, method) are not reported. 20-30% of the files that actually play a part in the development solution are recommended using Ying's approach [19].

2.4.2 ROSE — Zimmerman

Zimmerman *et al.* take a similar approach to helping developers navigate source code. Their solution is again to mine CVS to determine files that are modified together. Using this information they are then able to make recommendations to software developers about files that should possibly be modified based on their current work. Just like Ying's solution, because the system views all files in CVS, related resources that would otherwise not have been discovered through program analysis can be discovered. Files such as documentation can be recommended alongside source code files.

Again, much like Ying, Zimmerman observes changes within a sliding window (200 seconds in length) and groups all with matching author and commit rationale into a single transaction or atomic commit. Zimmerman chose to ignore all changes that affected more than 30 entities, unlike Ying's choice of 100 as an upper limit. These limits appear to be based on heuristic evaluation of what a 'meaningful' transaction entails.

Both Ying and Zimmerman along with many other solutions all mine CVS for their data. Unfortunately, there is a great deal of local interaction (file browsing, editing) that takes place between commits which is completely overlooked by these approaches.

2.4.3 ProjectWatcher — Schneider

To capture intermediate modifications, Schneider *et al.* developed a solution based on maintaining a local shadow CVS repository [21]. User edits are automatically committed to this shadow repository, increasing the granularity of information held related to user activity. This information is mined to provide Eclipse developers with an visual overview of the past and current modifications undertaken by their fellow developers. The goal of Schneider's work is to increase the developer's awareness of who is working on what code and when.

2.4.4 TUKAN — Schümmer

TUKAN is designed to keep developers aware of the activity of other developers. TUKAN was developed by Till Schümmer [7] and, much like Schneider's tool, is intended to help provide awareness cues to developers by mining local modifcation history. Schümmer introduces a distance model used to gauge how far away in software space your developer team members are working within the source code. Distance between artifacts is determined through a combination of structural (inheritance, composition) and temporal (creation/modification time) relations. These relations are then weighted with implicit or user defined weights to add strengths to each branch in the spatial model. Relations are updated dynamically as the users navigate TUKAN's spatial model. If a user moves between two methods, the strength of the corresponding relationship within the model increases, taking into account the user's familiarity with the surrounding code.

2.4.5 MYLAR — Kersten

A similar approach but based on a simpler model is Mylar. Presented by Kersten *et al.* [22], Mylar is a degree of interest (DOI) model for Eclipse inspired by the work of Card and Nation [23]. Rather than modeling 'distance' between elements, the DOI model holds interest values for each resource (coded as a single float). Selection of an element within the

Eclipse IDE increases that elements interest levels (+1 by default). Typing within a source document also increases the interest level (+0.1 by default). The most frequently used resources become apparent from their relatively large interest values. A decay mechanism is in place to reduce the importance of elements that have not been used recently.

MYLAR extends many standard Eclipse navigation views to leverage the DOI model. The DOI model acts as a filter, filtering unessential information from the Package Explorer, Outline view, Type Hierarchy, just to name a few. In addition, Mylar contributes a task view to Eclipse. The Mylar task view allows users to manage individual tasks, each representing discrete DOI models.

Initial studies suggest that this is a very effective approach. Their hypothesis was that if the task related resources are visible and highlighted in the standard navigation views, this will translate into less time seeking and more time working. To measure this the authors defined an edit ratio as "the number of keystrokes in the editor over the number of structured selections made in the editor and views." Baseline data was collected about the subject's Eclipse usage followed by use of Mylar enabled views. An average edit ratio improvement of 15% was seen [22]. The authors point out that this result may not be accurate due to task switching during the study. One user was reported to have increased their edit ratio by 49% [22]. This developer was working on the same project for the complete duration of the study whereas other subjects changed tasks at some point. This implies that this user saw a 49% increase in 'working' keystrokes over facilitation/navigation keystrokes.

2.4.6 Team Tracks — DeLine

Very similar to Mylar is Team Tracks by DeLine *et al.* [24]. The primary difference between Team Tracks and Mylar is that Team Tracks has social or collaborative filtering capability [17]. Team Tracks can integrate the navigation history of many developers when making recommendations and filtering views. The authors of Team Tracks establish the relatedness of items based on two intuitions:

- The more often a part of the code is visited, the more importance it has for someone new to the code.
- The more often two parts of the code are visited in succession, the more related they are.

A number of visualizations fall under the Team Tracks banner. These visualizations use the developer's navigation data to filter hierarchical information structures provided by the IDE (similar to Mylar). These visualizations are meant to "ease program comprehension by showing the source code navigation patterns of fellow development team members."[24]

Evaluation of Team Tracks showed stronger task completion numbers compared to subjects working without Team Tracks. Through a quiz, the authors of Team Tracks were able to show a significant improvement in program comprehension for those subjects who had Team Tracks installed [24].

2.4.7 FEAT — Robillard

A more specialized tool which can aid navigation is FEAT (Feature Exploration and Analysis Tool). FEAT, developed by Robillard and Murphy, provides a mechanism for identifying the features of a concern that is scattered throughout the software system (cross-cutting concerns) [25]. FEAT is based on a concern graph model, also by the Robillard and Murphy, that helps developers document and analyze structural dependencies within the code. Using this model the developer is able to navigate at a higher level of abstraction - a cross-cutting concern rather than raw source code.

FEAT requires manual intervention by the user when constructing concern graphs although recent work extends the manual approach. Although benefits will be reaped in future use of a mapped concern, from other research [1] we learnt that developers often work in a Just-In-Time fashion and that the benefits of such tools must be significant in order to justify the additional overhead required to produce results.

2.4.8 Hipikat — Cubranic

Another tool which helps navigation but whose goal is to help the newcomer "become productive more quickly" is Hipikat [26]. To accomplish its goals Hipikat makes a wide range of resources related to the user's current needs available. Hipikat continuously records and indexes all versions of the source code, bug reports, related archived communication (email) and web documents. Temporal proximity at check-in leads to association between resources much like the Ying and Zimmerman approaches discussed earlier. Relationships which are not explicit are inferred using keyword indexing. Hipikat recommendations are requested through selection of an artifact within the context menu. Recommendations are ranked and presented in a list within the Eclipse workbench.

The authors of Hipikat discovered that how useful a recommendation was depended on the context in which the recommendation was made and the knowledge/experience of the developer [26]. Developers new to a project may not have sufficient knowledge to interpret or gauge the suitability of recommendations made by Hipikat. For this reason the list of recommendations was augmented with a reason for each recommendation.

In addition, the authors observed that the quality and number of recommendations had an impact on the developer's performance. There is a substantial cognitive load associated with determining if a recommendation is relevant [9].

2.5 Discussion

The tools discussed above have different approaches to helping developers navigate. Some focus on awareness while other focus on determining all resources related to a particular task. Still others help form paths through the source code which can be more easily followed later (i.e., FEAT). We aim to target a specific form of navigation - the immediate or 'reflexive' level of navigation.

We frame our design rationale within the task characteristics identified by Marchionini as described in Section 2.1.

2.5.1 Specificity

We have chosen to target the near–immediate level navigation tasks characterized by revisiting familiar resources. These resources may temporarily not be identifiable through recollection. Momentary disorientation due to frequent digressions are one cause of this state. By providing recommendations which function as a *reminder* of related resources — developers may then recognize resources they had momentarily forgotten. This could help minimize the side effects of distraction and get the developer back on track more quickly.

But how do we provide these cognitive hints? From the literature review we learned that, in web navigation in particular, a great portion (58-81%) of navigation is comprised of revisiting previously seen web pages [27]. It has been observed that source code navigation is similar in many respects to navigating the web. Chalmers' web navigation path model covered in Section 2.2.4 leverages this effect to help users navigate the web. From these observations we develop the thesis that **developers' past navigation process may be useful in predicting their future navigation choices**. Where previous tools generate recommendations based on mining CVS commits, we focus on gathering fine grain navigation data and develop heuristics to produce recommendations. In principle this is similar to the approach taken by Mylar and Team Tracks. However, this approach differs in many ways. Recommendations are instead driven by the formation of explicit associations between files. Context of recommendation is based on the currently active file rather than a DOI model.

Based on analysis of user activity logs, Bannon *et al.* observed that it is important to be able to return the user to the exact state and location where they left off [12]. Since digressions are frequent when developing software, bringing the user back to a prior state is an important capability of their working environment. When opening files in Eclipse, the file is opened to the first line in the file. By retaining explicit associations between files and line numbers in other files we could possibly bring the user back to the specific location desired. Mylar achieves a similar effect at a larger scale. Mylar uses the task specific DOI models of activity to reveal different contents depending on the task.

A side effect of using line number resolution is that the tool becomes language *agnostic*, supporting development work within a heterogeneous environment. Many of the tools discussed in this chapter are tied to specific programming languages such as Java or Smalltalk. Documentation in HTML, source code in Java, scripts in Perl — all should be treated equally as each depends upon the other and will be of importance to the developer.

> "If users do not know the right questions to ask, they may miss important information." — Utting [28]

Although analytical search is a viable navigation tool, we are intentionally discarding this approach in favor of a browse based strategy as discussed in Section 2.1.1.1. Analytical search requires explicit input from the user and one of our design objectives is to minimize disruption of the user — analytical search is therefore not appropriate.

2.5.2 Volume

We aim to facilitate the developer's next step along their desired path. As such, the developer's expectation for necessary volume is minimal. If all I that is desired is to open File A, ideally only File A would be presented as an option. Though idealistic, this is a goal we work towards. Inevitably, as a recommendation tool, our current tool will provide a number of options. The usual strategy for dealing with many options is to form the information into a hierarchy. Hierarchies abound in the modern development environment and are one of the main locomotional structures used by browse–based interfaces. Unfortunately, hierarchies are often static and seldom aligned with the current task of the user (see Section 2.1.2). In addition, we have learned from both Marchionini and Jul, that any additional data presented to the user imparts additional cognitive overhead. This problem was reported by the Hipikat researchers as well. For this reason we chose a simple interface consisting of a short list (5 items or less) which are ranked as described in Chapter 3.

Tabs are a good example of a minimal interaction interface. Tabs within Eclipse serve the immediate need of returning to recently visited files. The interface is simple, requiring

a single click to reveal the desired resource. They work well when the file has been loaded recently, and the tab is still visible. Unfortunately, the tabs interface can quickly become cluttered when too many files are loaded at once. We see opportunity in providing an equally simple interface for developers, but populated with recommendations related to the *currently active file*. By providing a simple short list of recommendations, the interface is visible and easy to use. This has the potential to support immediate finding of resources that perhaps have been forgotten but are recognized when seen in the short list.

The accuracy required is such that the recommendations include the file to which the user is most likely to navigate. Many recommendation systems and search engines (e.g., Google) provide results that are not 100% accurate but are very efficient (presented rapidly). They are seen as useful and are accepted by most users. We propose that a recommendation based tool for navigating software with similar levels of efficiency and accuracy will also be useful. Obviously if its accuracy is below a certain threshold, the tool will be rejected. However, we believe that accuracy does not have to be perfect for recommendation based tools to be accepted and used.

2.5.3 Timeliness

The user's expectations for immediacy automatically imply the need to produce results quickly and that the results must be current enough in order to be useable. Therefore, we see advantage in developing a self sufficient solution: immediate responsiveness, current (up-to-date), and ease of adoption.

The tools discussed previously have different approaches to determining what resources will best support the task which ultimately determines their expediency. Many of these systems rely on analyzing data in CVS and other external resources. Although these resources have a great deal of information, it takes time to process that data, it is potentially stale, and it is subject to network failures. By remaining self sufficient (relying only on data generated by the tool itself and not on external data sources) these potential performance bottlenecks can be minimized.

A positive side effect of being self sufficient is the elimination of additional infrastructure requirements. The tool is immediately usable making it easy to integrate and adopt as a potential solution.

2.5.4 Summary

To summarize, the following is a list of the design objectives that emerge from the preceding discussion.

- Form recommendations based on previous interaction history (movement between editors within Eclipse)
- Provide a lightweight, self-sufficient, easily adoptable solution
- Present user with short pick list of recommendations (approx. 5 recommendations)
- Tool must be responsive if we are to meet the expectations of timeliness
- Contextual recommendations based on currently selected file
- Selected recommendation opens file to contextually relevant location within the file (instead of opening to the first line)
- Language agnostic (works with all text based files)

In the following chapter we will explore these design objectives though implementation of a prototype tool for navigating software.

Chapter 3

Design and Implementation

> "This is an example of a chicken-or-egg problem of user-centered design: We cannot discover how users can best work with systems until the systems are built, yet we should build systems based on knowledge of users and how they work. This is a user-centered design paradox. The solution has been to design iteratively, conducting usability studies of prototypes and revising the system over time..." — Marchionini [9]

When designing a tool, it is desirable to have an understanding of the task for which you are designing. The previous chapter started with a theoretical perspective of the navigation task followed by a survey of current navigation tools. Consideration of this background led to a number of features which we test in prototype form. This chapter details the design and implementation of this prototype — NavTracks.

NavTracks offers developers recommendations for related files given their previous navigation patterns. Figure 3.1 shows the NavTracks Related Files view within the Eclipse IDE. To the right side of the figure is the file that the developer is currently viewing, the active file. Below the Related Files view is the Package Explorer view, which is the Eclipse default file and folder browsing tool. The Package Explorer allows a developer to navigate via the hierarchical containment relationships defined for the system. NavTracks is implemented as a complementary tool to the Package Explorer. The three files shown in the Related Files view are related to the active file in terms of navigation history. The files are ranked so that the file highest in the list is the most recently formed association to the

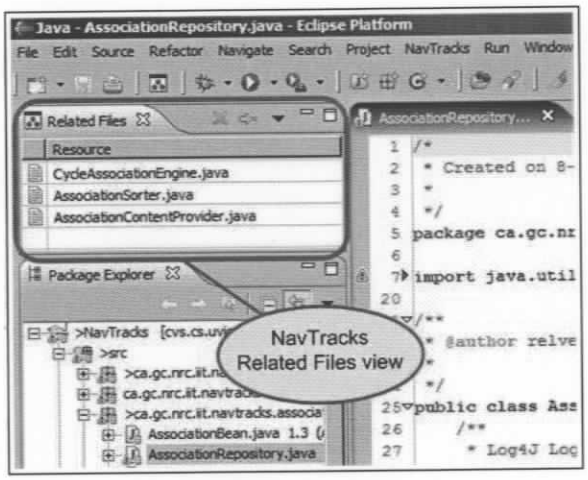

Figure 3.1. *NavTracks Related Files view*

currently active file. If a developer clicks on one of the files in the Related Files view, the clicked-upon file will open in place of the currently active file. When the clicked-upon file is opened from the Related Files view, the cursor will be placed at the location in the file where it was last located, and the Related Files view will be updated to reflect the associations for the newly opened file.

Associations are created via a three-step process. First, each file selection is collected into an event stream. The event stream is then filtered to remove redundant navigation events. Finally, the event stream is examined for possible associations which are stored in a repository.

3.1 Events

The collection of data is implemented as a listener on a developer's workstation. Each time the developer views a file, either from opening the file or selecting it from a tab, a reference to the file is placed into the event stream. Metadata is stored with the file reference and is discussed later in this section.

As the event stream is constructed, redundant events are filtered. Currently two types of

events are removed: jitters and duplicates. Jitters occur when a developer moves between files very quickly. Currently, the threshold for jitter events is one second. If a developer spends less than one second in a file, the reference to the file is not added to the event stream, as we assume that this was not a meaningful navigational event. We also remove duplicate events from the event stream. If the same file is referenced twice sequentially, we assume that this was a navigational error, and filter the event.

We recognize that duplicates may not represent navigational errors and that the one second threshold may be too high or too low. More study of the developers' navigation patterns is required to evaluate the appropriateness of these design decisions. If these filters turn out to be inaccurate they can easily be refined. Moreover, new filters can be added.

3.2 Associations

The algorithm responsible for forming associations is based on our observation that files, which participate in short navigation cycles, tend to be related. A cycle is defined as a series of file accesses by the developer, beginning and ending with the same file. For example, if the developer accesses File A, then C, then B, then A, NavTracks records an ACBA cycle. Upon detection of a cycle, the algorithm forms associations between the first file in the cycle and each file contained within the cycle. This is illustrated in Figure 3.2 and described below.

Events in the event stream are passed through an event window of size n. As events arrive, a cycle detect algorithm searches for cycles within the event window. In the current prototype, an event window of size n = 4 is used, and a maximum cycle length (number of edges) of size k = 3. A window size of 4 was chosen based on the observation that longer navigation paths have a greater potential to contain extraneous files. Dynamically adjusting the size of the window depending on the number of open editors has been considered as a possible improvement to the current implementation.

Steps 1 - 5 in Figure 3.2 show how associations are formed. Each letter represents a

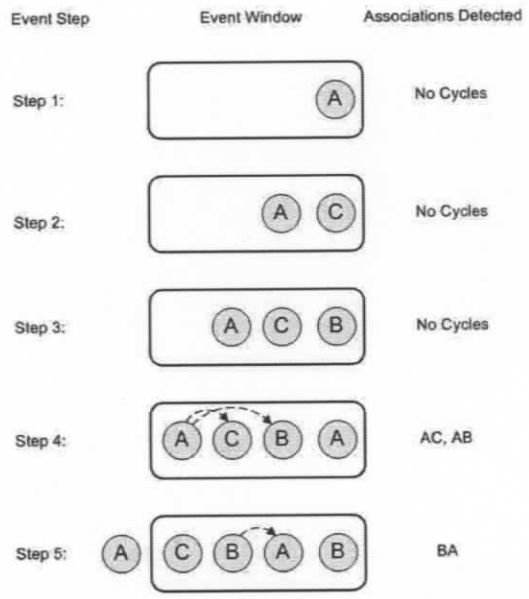

Figure 3.2. *Associations formed via cycle detection, event window size is n=4*

unique file reference in the event stream. Conceptually, events enter the window on the right and exit to the left. Step 1 shows the first navigation Event A entering the event window. As the next event occurs, A shifts to the left making room for Event C (Step 2). In step 3, the events in the window are shifted to the left once more when Event B arrives. At this point the events in the window are A-C-B. As no cycle is detected, no associations are formed up to this point. Step 4 shows a second Event A entering the event window, resulting in the detection of a cycle of size 3, A-C-B-A. Upon detection of this cycle, associations are formed between File A and each file contained within the cycle. This results in associations AC and AB being constructed. At Step 5, Event A exits the window to the left and event B enters from the right. Here a cycle of size 2 is detected — B-A-B. The association BA is then constructed. Pseudocode describing this algorithm is provided in Appendix A.

Note that the associations are not commutative. That is the association AB is not equal to the association BA. A strong association from A to B under certain circumstances does not necessarily imply a strong relationship from B to A.

Once formed, associations are stored in an association repository. Associations are

stored with metadata indicating the frequency of the association, along with the line last visited and the time of occurrence. If the association already exists in the association repository, the line number and time stamp metadata are updated to reflect the reoccurrence, and the frequency metadata is incremented.

This continuous integration of association data eliminates the need for batch processing of navigation logs or CVS data. As discussed in Section 2.5.3 this is intended to keep the interface responsive and current.

3.3 Recommendations

Recommendations are made based on the data in the association repository. Each time a file is opened or navigated to, the Related Files view initiates a four step process to display the related files. First, all associations with the currently selected file as its antecedent are retrieved. Second, the associations are screened for inconsistencies with the local project. That is, if a file is recommended that no longer exists in the project (a file that has been recently deleted or renamed), then this recommendation is not displayed. Third, the associations are ranked in descending order based on time of occurrence. Although available in the association repository, association frequency was not used for ranking. This is because when the developer is working on several tasks at once, or switches to a new task, frequency is not a good predictor of sought-after files. That is, a file may have recently been accessed frequently, but because of a task change, is no longer relevant. If we ranked based on frequency, the new relevant files may not have ranked high enough to be shown in the Related Files view. Currently, we save frequency so that in the future, we can explore alternative ranking algorithms, perhaps combining frequency and time information. Finally, the recommended file names are displayed in ranked order, more recently occurring files appearing higher in the list.

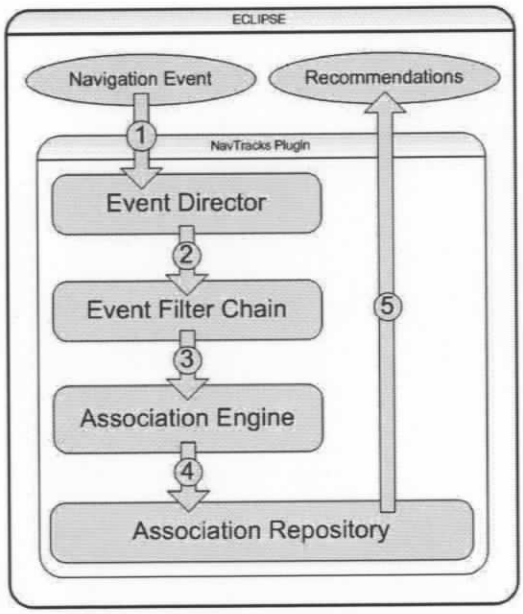

Figure 3.3. *NavTracks architecture*

3.4 Eclipse Integration

NavTracks is built on the Eclipse platform. Figure 3.3 shows an abstract representation of the NavTracks architecture. The arrows represent the flow of information through NavTracks — associations being formed (1 - 4) and recommendations displayed (5). The architecture is modular so that we and other researchers or developers can easily adapt and extend NavTracks.

The association engine and metadata collected are completely replaceable. The size of the event window and the length of cycles detected are customizable. Additionally, extra filters can be added to prune the event stream. Moreover, because NavTracks' event stream collects file selection events indifferent to file type, it can capture the relationship between all types of textual files in a project. For instance, NavTracks can form associations between Java, XML, PERL, and HTML files, thus opening the possibility of automatically associating documentation and code.

The Association Repository is an abstraction which wraps Eclipse's native *IMarker*

interface. Consequent file identification and all association metadata is stored within a custom marker tied to the association's antecedent file. This makes the generation of recommendations very fast since files of interest get 'marked' on the antecedent file.

The recommendations are accessible in the Related Files view (Figure 3.1). The Related Files view is implemented as a simple list view that integrates with the Eclipse Java Perspective. Upon opening a new Java Perspective, the Related Files view is positioned directly above the Package Explorer.

3.5 Summary

We have described the design and implementation of a light weight navigation aid for Eclipse. The tool provides a short list of recommendations to the developer which may be quickly glanced at to determine if the desired resource has been presented. The tool contributes a simple list based view to Eclipse which is populated through examination of historical navigation processes.

To learn from this prototype and direct future design decisions we evaluated the tool using real data from real world developers. The following chapter details the quantitative and qualitative evaluation of NavTracks.

Chapter 4

Evaluation

NavTracks has two major facets, a user interface (the Related Files view) and the back-end that generates recommendations for display. The algorithm used in the back end (described in Chapter 3) determines what recommendations get propagated to the user interface. For evaluation, we are interested in how well this algorithm predicts the users subsequent navigation. Second, we are interested in how users respond to the Related Files view. Do they find it useful? We begin our evaluation with a quantitative analysis of the algorithm currently used in NavTracks followed by the qualitative reports from subjects which participated in a small study.

4.1 Quantitative Evaluation: Algorithm Assessment

4.1.1 Gathering Test Data

Our evaluation process is based on concurrent evaluation and training over sample navigation event sequences. Sample navigation event sequences were gathered from five participating subjects. Each sequence consists of navigation events generated from user file selections within the Eclipse IDE (Integrated Development Environment).

File selection events are generated from many sources. File selection events can be generated by selecting a file from within the Package Explorer, selection of an inactive Tab, and pressing the back button navigation button just to name a few. Any event that

causes a file to receive focus in Eclipse is recorded. Along with the location and name of the file being opened, the time at which the event occurred is also recorded.

Upon obtaining ethics approval, the navigation data of five developers was collected for use in the evaluation process. Table 4.1 summarizes each subject's programming experience, software system and task description.

Table 4.1. *Algorithm performance analysis users*

User ID	Programming Experience	Software System	Task Description
1	Prof.Developer, 5+ yrs	Java App	Development/Maintenance
2	CO-OP Developer	Java App	Maintenance
3	CO-OP Developer	Small Java Apps	Development
4	CS Phd Student	Java App	Development
5	Prof.Developer, Recent Grad	Small Java Apps	Development/Maintenance

4.1.2 Filtering

The data is subjected to the filters as described in the preceding chapter. First, the duplicates filter removes identical events which occur side-by-side sequentially in the event stream. Second, any quick hops, or *jitter*, between files that occur in less than one second are removed.

4.1.3 NavTracks Assessment

To assess the accuracy of the algorithm used in NavTracks we compare the predictions made by NavTracks with the actual events as they occurred in a subject's event stream. After discussing the results of this process we apply the same process (tooling and data sets) to an alternate algorithm in order to do a comparative evaluation.

To help assess the algorithm's ability to produce valuable recommendations, developers' file navigation events are used to train and test the algorithm. The event streams are filtered for duplicates and jitter, then used as the basis for assessing the accuracy of the algorithm. The assessment process consists of applying the NavTracks algorithm to each

event in the stream and recording the algorithm's success or failure to predict each subsequent event (file). As each event is evaluated, it is also used to train NavTracks. Thus, a continuous evaluation and training process occurs over the entire event stream.

For each event fed into the algorithm, a set of recommendations is produced by NavTracks. The set of recommendations is predicated on the current event (file). If this is a first occurrence of this event type (first time opening this file) then there will be no recommendations made. This is true of the current implementation. Alternate implementations which would handle this situation better are discussed in Section 5. As the number of observed events increases, the number of recommendations will increase. The total number of recommendations is capped at five for reasons discussed in Chapter 3. This set of recommendations, empty or not, is used to gauge the accuracy of the algorithm.

Accuracy is determined by calculating the percentage of correct predictions out of the total number of events in the event stream. This process consists of comparing each subsequent event in the event stream to the previously generated recommendations. The recommendations are actually an ordered list from Rank 1 to Rank 5. Rank 1 is displayed at the highest point in the NavTracks Related Files list view. This rank represents the strongest confidence level — the file we most expect the user to require. The current implementation designates the highest rank to the most recently formed association. The file in each rank is compared to the current event. If a match is found, a hit, the total number of successful predictions is incremented and the rank at which the match was found is recorded. If the file is not found in the list of recommendations, a miss, the number of failures is incremented. Accuracy of the algorithm is calculated as the number of hits divided by the total number of hits and misses (number of events in the filtered event stream). The results of this process are summarized in Table 4.1.

Based on the five navigation streams collected, the cumulative accuracy of the Cycle Detect (windows size = 4) algorithm is 39%. This means that 39% of the time NavTracks would have made a recommendation that corresponded to the navigation path chosen by the developers. This 39% is distributed among the ranks as seen in Figure 4.2. We can

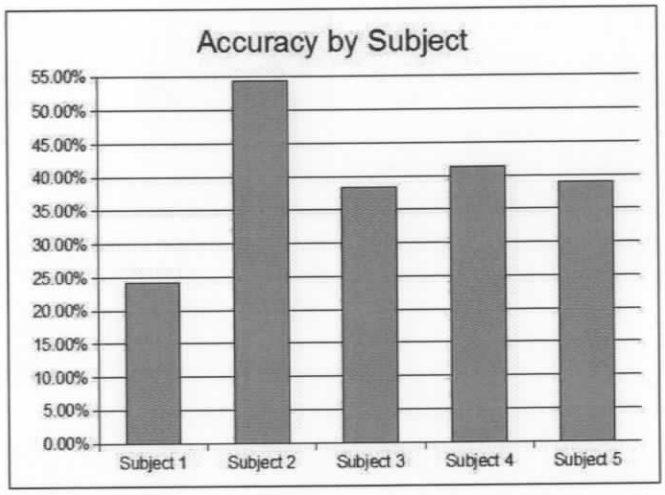

Figure 4.1. *Accuracy by User, event window size is n=4*

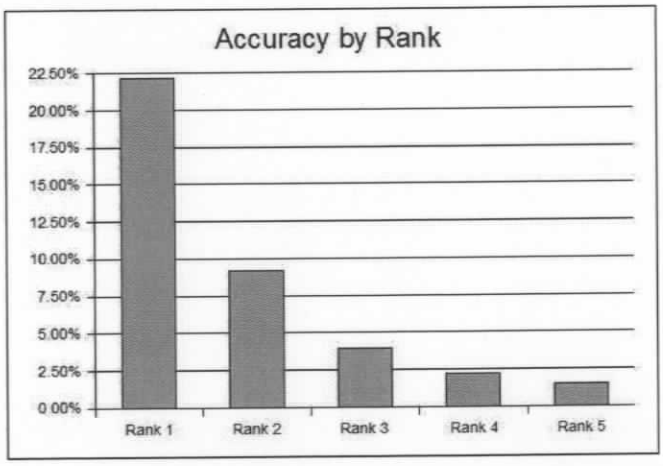

Figure 4.2. *Total Accuracy by Rank, event window size is n=4*

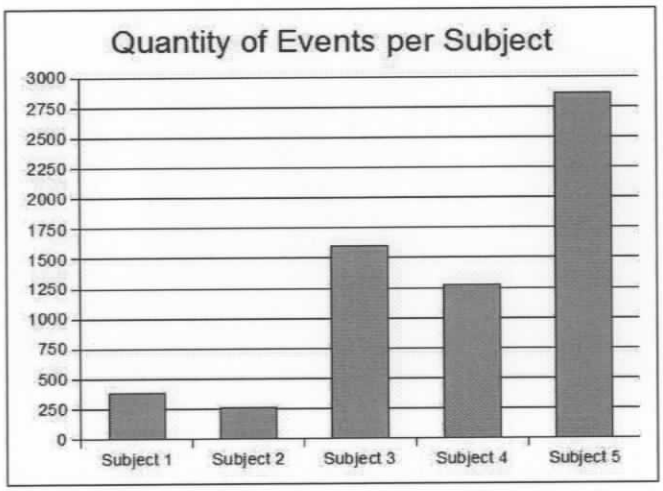

Figure 4.3. *Quantity of Events per User*

see that accuracy tapers off significantly after Rank 1. Rank 1 holds the greatest portion of correct predictions at 22%. The following four ranks taper off significantly.

Table 4.1 shows a significantly lower accuracy for Subject 1. In Figure 4.3 you can see that Subject 2's quantity of events is also low but accuracy is very high (Figure 4.1). From this we can see that the significate accuracy difference is not the result of the low volume of data provided by Subject 1. A more probable explanation for Subject 1's relatively low accuracy is the subject's percentage of unique events. The more unique events that occur, the less chances to make correct predictions. Figure 4.4 shows that Subject 1 has a relatively larger proportion of unique events then the other subjects.

The above process consisted of a 100% training and evaluation over the data set. A comparison of these results to a 50% training 50% evaluation scheme was also conducted. The expectation was that this would increase the accuracy levels in each rank. However, this was not the case. The difference was negligible between these two processes. This is perhaps due to recency being the primary determinant of rank in the current implementation. If a strong association was formed in the first 50% and it's antecedent visited in the last 50%, only the most recent are displayed which may not include the formerly constructed strong relationship. This illustrates a weakness in the current approach and is the

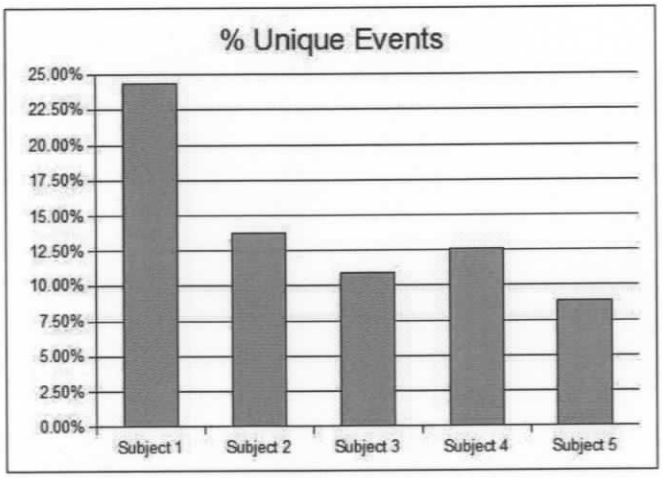

Figure 4.4. *Percentage of Unique Events*

subject of future work.

To get a better idea of the accuracy of the cycle detection algorithm, we did a comparison between the algorithm and a modified version of the stack based navigation history found in Eclipse. This modified history stack is a FILO (First In Last Out) stack **with duplicates removed**. The traditional History Stack provides redundant, repetitive data which would result in irrelevant recommendations. By removing duplicates, we provide the most recent, *unique* events. This has the potential to increase the accuracy of the recommendations.

Figure 4.5 shows how this stack operates for the sequence of events 1, 2, 3, 2. As each event occurs, any matching events (duplicates) within the stack are removed. Second, the item is pushed onto the top of the stack. When event 2 occurs for the second time, its previous occurrence is removed and the new occurrence is pushed on the top of the stack. In this way, the most recently accessed file is always at the top of the stack.

Figure 4.6 shows the accuracy by rank when using the history stack compared to cycle detect. To produce these results, a history stack as described above was used as a recommendation algorithm similar to cycle detect. This new configuration was subject to the same input data and filter process. In similar fashion to cycle detect, only the top five events

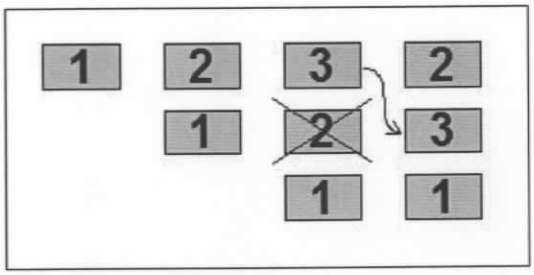

Figure 4.5. *History Stack Example*

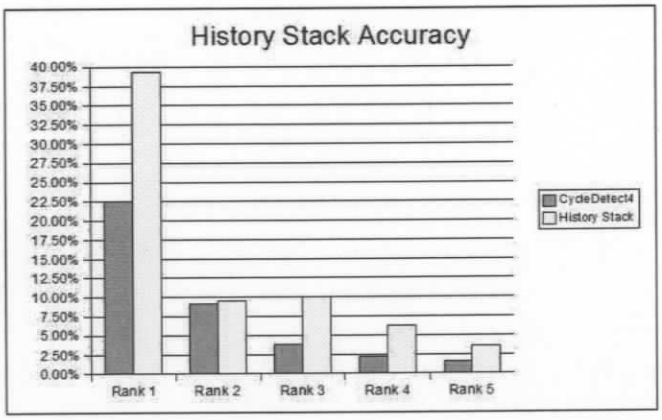

Figure 4.6. *History Stack vs. Cycle Detect*

are used as recommendations - the five most recently accessed files in the case of the History Stack. As can be seen from the chart, history stack outperforms cycle detection when predicting future navigation decisions.

Clearly, there is room for improvement in the performance of the NavTracks algorithm. This being said, a larger study consisting of more navigation data would improve our confidence in these findings. Opportunities for improvement are discussed in Chapter 5.

4.2 Qualitative Evaluation: User Feedback

In addition to the above quantitative evaluation, five additional users were asked to try NavTracks. Three of the users were expert developers, two of which had at least 5 years of experience. One user was a co-op student and the other a graduate student. Three users

were maintaining large systems with >5 KLOC and two users were working on systems with less than 5 KLOC. Table 4.2 summarizes the skill level and tasks being undertaken by these subjects.

Table 4.2. *Observed users*

User ID	Programming Experience	Software System	Task Description
6	Graduate Student	Java App	Development
7	Professional, 5+ yrs	Small Java Apps	Development
8	CO-OP Developer	JSP Web App	Major Refactoring
9	Professional, 5+ yrs	Perl Web App	Dev/Main-tenance
10	Prof. Developer, 2+ yrs	Java App	Maintenance/ Understanding

From observation and discussion with the above users, three navigation personalities emerged: Newcomer, Wanderer, and Navigator.

4.2.1 Newcomer

User 8 was a newcomer to an ongoing development project. This developer was unfamiliar with the source code — not knowing where specific code was located within the package hierarchy. As the developer was unfamiliar with the code, it took a considerable amount of time and effort to find the files related to a specific maintenance task. Specifically, he was having great difficulty remembering the names and locations of related files when returning to work on a previously visited area of the code. Similarly to this user, User 4 was just starting to develop a new system. He had to navigate throughout the package hierarchy to find the files and classes that he wanted to incorporate into his system. This navigational task was taking a considerable amount of time.

NavTracks helped both of these developers by providing a list of recommended files in the Related Files view. Whereas they were having trouble recalling the names and locations of files, they could easily recognize them when they saw them. Furthermore, using NavTracks, these developers were able to navigate directly to the files, rather than having to hunt through the Package Explorer. For these developers, NavTracks aided in navigation by providing a memory aid for related files.

4.2.2 Wanderer

User 6 did not have such a successful NavTracks experience. In fact, he reported no valuable recommendations arising from NavTracks usage. Through an interview with this developer, it was determined that this may have been due to the type of work being completed. The developer was involved in a major refactoring of a web application written using Java Server Pages. The work involved a repetitive copy, paste, and modify cycle that did not involve returning to previously visited code. Because of this, the developer did not receive any meaningful recommendations from NavTracks. NavTracks only works well when a developer revisits previously viewed files.

4.2.3 Navigator

User 7 was involved in a project that required frequent reference to a specific central file. He found NavTracks particularly useful because when he used the Related Files view to go back to the central file, he would be placed at a contextually relevant section of the central file because NavTracks remembers the last line visited. As an illustration, when he went back to the central file from File A, he would be placed at a different line number than when he went back to the central file from File B. This greatly simplified his task because he was not left searching the central file for the relevant section. This subject also frequently used NavTracks as a stand-in for a recently visited file list. Because he was working on a relatively small system, where there were large interdependencies between files, NavTracks allowed him to see which files he was recently looking at. He found this to be more useful than the tab structure of the Eclipse system, which truncates file names, and does not show all files on the screen when a large number of files are open.

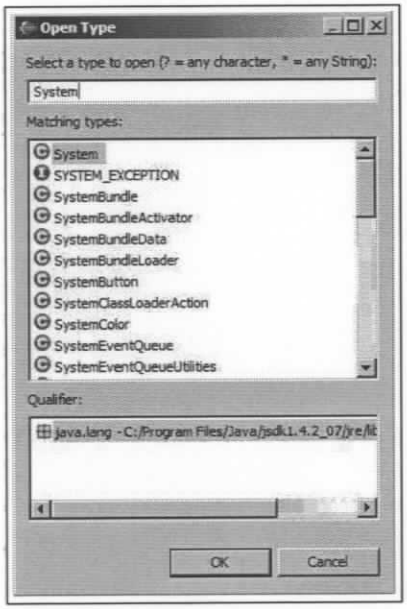

Figure 4.7. *Open Type Dialog*

4.3 Discussion

We set out to achieve high levels of specificity, volume and timeliness when designing NavTracks. Results of evaluation showed an average of 39% accuracy based on the collected data streams. This shows a great deal of promise and later in this section we discuss possible algorithm alternatives that may increase the accuracy of NavTracks.

Users reported using NavTracks infrequently. Informed by navigation literature we intentionally limited the volume of recommendations displayed within the Related Files view. Although this may have reduced cognitive strain in terms of evaluating each individual recommendation, user feedback indicated that the amount of screen real estate taken up by this list was disruptive. Comfort and familiarity with existing navigation aids was a common reason cited for not using NavTracks. Experts familiar with the naming conventions used in the source code tended to use the Open Type dialog exclusively (see Figure 4.7).

Users did however report usage of the Related Files view (Newcomer, Navigator). Features of particular interest to users were NavTracks' ability to work with different file types

and open files to contextually relevant locations within files.

In terms of timeliness, NavTracks' self sufficient design produced the desired effect. Recommendations were instantly presented in the Related Files view the moment a file selection was made. Perhaps the combination of NavTracks' current efficiency but presented in pop up form via a simply key sequence would improve adoption.

4.3.1 Future Experimental Considerations

The results of this evaluation are based on a limited study involving ten software developers. These results would be more authoritative if we increased the number of subjects and the amount of navigational data collected from each subject. In addition, a task specific evaluation in which all subjects complete the same task could produce interesting results.

Although the History Stack data was used to evaluate NavTracks, it says nothing about the actual use of the history mechanism found in Eclipse. How often is it used? How frequently are items in the back button drop down list selected? Unlike Eclpise's back button, NavTracks' Related Files view immediately exposes recommendations in a list form. The back button drop down list is hidden prior to selection. Does the additional effort required to expose the back button list hinder usability? The above questions, that when answered, may help improve future navigational design.

4.3.2 Algorithm Considerations

To see if the window size of the cycle detection algorithm would make a difference, we ran the same test using a window size of 8. Figure 4.8 shows the results of cycle detection with a window size of 8 alongside the window size 4 results and the history stack results.

You can see that by doubling the event window size we only get a marginal increase in accuracy. With a window of size 8 the rank 2 accuracy matches that of the History Stack but for all other ranks, History Stack still performs better.

Additional variations on this theme could be conducted including forming bi-directional

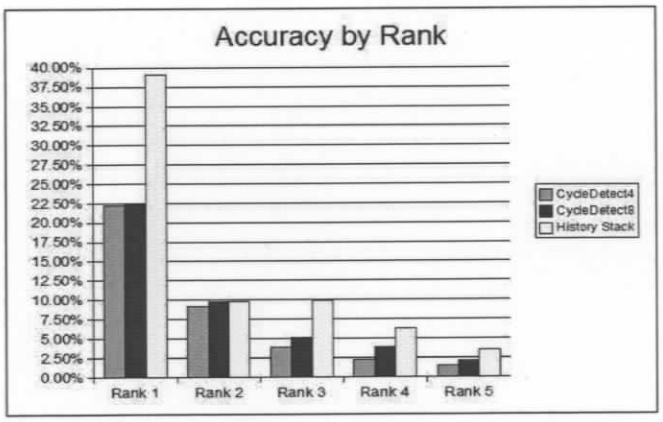

Figure 4.8. *Cycle Detect 4, 8, and History Stack*

associations. However, using both recency and frequency in the ranking algorithm may be more profitable in improving overall accuracy. This will be discussed in future work (section 5).

4.3.3 Usefulness

Are the recommendations made by History Stack useful? Even though it is superior to Cycle Detection, the results are still rather low with the exception of Rank 1. How useful is the data in the remaining 4 ranks? Unfortunately, we do not have enough user data to make any reasonable statement concerning the utility of these results. What our data does highlight is that a seemingly large portion (in some cases approaching 50%) of the developer's activity involves repeatedly moving back and forth between two files. This suggests a point of interest for future research.

4.3.4 Adoption

How does NavTracks compare to similar tools in terms of ease of adoption? Certainly one measure of a tool's success is how well it is adopted by its target users. NavTracks was designed with simplicity in mind. NavTracks' implementation as a plugin for Eclipse

makes it immediately available to all platforms based on the Eclipse platform. For this reason installation is simple, easing adoption. In addition, by not relying on external services, once installed, NavTracks' features are immediately useable without additional configuration overhead. Tools that require maintaining an external database system or connection to external code repositories immediately force configuration issues onto the user, potentially hampering tool adoption.

As we mentioned earlier, enhancement to the user interface may improve the adoption of NavTracks. In the following chapter we will take a closer look at potential future directions for this research.

Chapter 5

Future Work

In this chapter we propose avenues of future research based on the results of the present work.

5.1 Algorithm

It would be interesting to do a comparison of the NavTracks algorithm with those used by some of the tools mentioned in the background. Future work could entail implementing algorithms such as the Chalmers' Path Model and doing a comparison. This comparison would be interesting as Chalmers' method mines a complete database of events removing recently occurring events from the set of recommendations. NavTracks does not remove recently occurring events from the recommendations nor does it mine frequent 'paths'. This comparison would help in understanding if the overhead of these additional capabilities would be justified in the next iteration of NavTracks.

Another potential avenue to explore would be augmenting the current algorithm with frequency. Currently rank is determined by recency. Frequency of occurrence is not being taken into consideration. Future work may entail consideration of both time and frequency in determining rank. An interesting approach to this would be to augment the current system with input from the Mylar DOI model. Before presenting recommendations in the Related Files view, NavTracks could query the Mylar DOI to determine if the recommendations are not justified based on the current task context. This has potential to increase the

accuracy dramatically.

Another approach to increasing the accuracy of NavTracks would be to form bi-directional associations between resources rather than uni-directional. Currently associations are uni-directional. For the cycle ABCA, the associations formed are AB and AC. If bi-directional arcs were formed, the same cycle would produce the associations AB, AC, BA, and CA. Although this infers associations that are not explicit in the event stream, this may lead to higher accuracy levels.

5.2 User Interface

5.2.1 Tooltips

In terms of improving the current interface, the Related Files view could be augmented with small code snapshots or summaries which are revealed upon mouse-over. Dumais shows that inline summaries in search results helped users [29]. Perhaps tool tip style summaries which include a summary of the code surrounding the location would be useful.

5.2.2 Explicit Interaction

This work used recommendation techniques to target users' immediate ephemeral navigation needs. The evaluation results suggest that, as implemented, this is not a viable approach. In future work it may be necessary to step back from the implicit (recommendation) approach and focus on explicit user interaction. However, user feedback suggests that the data supplied by NavTracks was at times accurate and useful.

Often users suggested a need for more control over the information being presented. Perhaps giving the user the option of explicit control augmented with recommendations would be a better approach. When moving and making decisions quickly, any extraneous information breaks user concentration and reduces efficiency. The qualitative results of the user study (in particular the 'Navigator') suggest that current tools fall short of providing

the user with sufficient ephemeral navigation support. Tabs, as discussed in Section 2.3, do have merit but perhaps improving upon this interface should be the focus of future work. Maintaining visibility of pointers to specific locations in the source, flexibility to organize these pointers into any order deemed suitable by the user, these are capabilities that are lacking and could be the target of future research.

5.3 Visualization

There may be value in the application of visualization to aid in recovering the 'implicit' architecture of the system. What is meant by implicit architecture is how people move about in a system, as opposed to hierarchical or semantic relationships that can be used to define system architecture. Future work may wish to address whether this notion of implicit architecture may aid in program comprehension and general maintenance activities including impact analysis. The current list style interface employed by NavTracks does not provide the user with sufficient orientation queues. This was one of the Cognitive Design Elements suggested by Storey *et al.* [4].

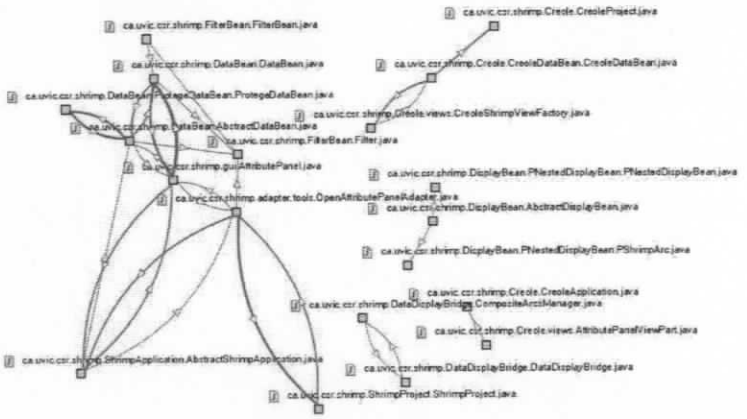

Figure 5.1. *NavTracks Visualization*

Figure 5.1 shows a visualization of the NavTracks cycles detected during one programming session of an expert programmer. Arcs between files represent an association, with

thickness representing the strength of the association (the number of times it was detected). The visualization was created by extending the Creole visualization plugin for Eclipse [30].

The visualization represents changes to Creole by a Creole developer. Upon seeing the visualization, the developer was surprised by the results. Some expected arcs were absent and yet others were thicker than expected. This may indicate that this perspective of the code may provide information beyond that simply available by looking at strictly structural relationships.

5.4 Collaboration

Support for collaboration centers around the sharing of path information. Both Wexelblat [5] and Chalmers [7] conceived interaction patterns as providing the basis for sharing information in a community. The interaction tracks that we discover for a software space may be useful if used in a community context. There are several things we could implement to do this. First, we could save the tracks of experts as they browse the software space. This information could be useful to others trying to understand hidden dependencies in the software. Second, we could save tracks as they relate to a particular task or context (à la Mylar). Then when others are faced with the same or similar task or context, they could access the tracks. Finally, we could combine tracks from several users to form a larger model of the relationships between files. This approach is taken by Deline *et al.* in their recent work [24]. By combining developer's tracks, we may get a broader view of the system, and the additional data may help to uncover more valid tracks (in the sense that additional data provides additional support for particular relationships). With respect to sharing tracks, it is interesting to note that Wexelblat found that navigation on the web using tracks was mostly useful for those already familiar with that information space. It may be that recommendations will not be useful for software developers who do not already have a good conceptual model of the software space. This requires more investigation.

Chapter 6

Conclusion

This research has looked at the domain of physical and electronic navigation and considered its relevance and application to the domain of software engineering. We have highlighted the implication of task specificity, timeliness, and volume to navigation tool design. In addition, this work presents a number of valuable contributions:

Background The Background chapter provides future researchers with a survey of existing navigation tool designs. This, along with the bibliography will help future navigation design researchers identify related background research.

A Recommendation based Tool As part of this work we presented NavTracks — a tool for helping developers navigate source code. Its recommendation style interface was populated through detection of navigation cycles within the Eclipse IDE. Results of initial evaluation suggest that short cycles in navigation may not be a good predictor of future navigation choices. Although this may not be conclusive, future work has been proposed that may yet produce desirable results and more conclusive evidence.

Flexible Architecture for Navigation Experimentation NavTracks was designed with ease of navigation experimentation in mind. It handles event collection, filtering, association forming, and persistence. The Filter Chain allows the addition and removal of filters that alter which events may pass or which are blocked. The Association Engine can be completely replaced in order to change how associations are formed. Finally, the Association Repository can be swapped out to change how associations are persisted. This all combines to form a very flexible framework upon which to

experiment.

Need for ephemeral navigation support within Eclipse One unexpected result of this research is the increased awareness of the need for better ephemeral navigation support within Eclipse. Two observations motivate this conclusion. First, the frequency of movement between two files was significant in the data provided by software developers. This suggests that tooling for fast switching between two software artifacts may require further attention in future IDE navigation design. Second, comments from subjects regarding a desire to have control over what files remain visible in the Related Files view suggests a need for some measure of explicit control over temporarily desirable resources.

Navigation of source code is an exciting area of research with a promising future. The present research only begins to make steps towards better navigation design in software development environments. Through the course of this research numerous problems rooted in navigation have been identified that negatively affect the software development experience. However, solutions to these problems are beginning to emerge out of collaboration between many different disciplines including Human Computer Interaction, Information Retrieval, Data Mining, Psychology, and Software Engineering itself. The results of this collaboration promise innovative solutions that will give software developers the tools they need to develop software more comfortably and efficiently.

Bibliography

[1] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97. IBM Press, 1997, p. 21.

[2] J. Araujo, A. Moreira, I. Brito, and A. Rashid, "Aspect-Oriented requirements with UML," in *Proceedings of the 5th International Conference on the Unified Modeling Language and its Applications*, Dresden, Germany, 2002.

[3] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. Cox, "Browsing and searching software architectures," in *15th IEEE International Conference on Software Maintenance*. ICSM'99, 1999, pp. 381–390. [Online]. Available: citeseer.ist.psu.edu/sim99browsing.html

[4] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," in *International Workshop on Program Comprehension*, 1997, pp. 17–28. [Online]. Available: citeseer.ist.psu.edu/storey97cognitive.html

[5] K. Scheiter, P. Gerjets, and E. Heise, "Hypertext navigation and conflicting goal intentions: Using log files to study distraction and volitional protection in learning and problem solving." [Online]. Available: citeseer.ist.psu.edu/scheiter00hypertext.html

[6] J. Park and J. Kim, "Effects of contextual navigation aids on browsing diverse web systems," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00. New York, NY, USA: ACM Press, 2000, pp. 257–264.

[7] T. Schümmer, "Lost and found in software space," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*. Washington, DC, USA: IEEE Computer Society, 2001, p. 9066.

[8] D. Janzen and K. D. Volder, "Navigating and querying code without getting lost," in *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03. New York, NY, USA: ACM Press, 2003, pp. 178–187.

[9] G. Marchionini, *Information Seeking in Electronic Environments (Cambridge Series on Human-Computer Interaction)*. Cambridge University Press, March 1997.

[10] S. Jul, "From brains to branch points: Cognitive constraints in navigational design,"

Ph.D. dissertation, University of Michigan, Computer Science and Enginering, 2004. [Online]. Available: http://www-personal.umich.edu/~sjul/research/

[11] J. Conklin, "Hypertext: an introduction and survey," *Computer*, vol. 20, no. 9, pp. 17–41, 1987. [Online]. Available: http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=29486#

[12] L. Bannon, A. Cypher, S. Greenspan, and M. L. Monty, "Evaluation and analysis of users' activity organization," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '83. New York, NY, USA: ACM Press, 1983, pp. 54–57. [Online]. Available: http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=801580#

[13] V. Kaptelinin, "UMEA: Translating Interaction Histories into Project Contexts," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '03. New York, NY, USA: ACM Press, 2003, pp. 353–360.

[14] J. Budzik and K. J. Hammond, "User Interactions with Everyday Applications as Context for Just-in-time Information Access," in *Proceedings of the 5th international conference on Intelligent user interfaces*. New York, NY, USA: ACM Press, 2000, pp. 44–51.

[15] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker, "TaskTracer: A Desktop Environment to Support Multi-tasking Knowledge Workers," in *Proceedings of the 10th international conference on Intelligent user interfaces*, IUI '05. New York, NY, USA: ACM Press, 2005, pp. 75–82.

[16] M. Chalmers, K. Rodden, and D. Brodbeck, "The Order of Things: Activity-centred Information Access." Brisbane, AUS: International Conference on the World Wide Web, 1998. [Online]. Available: http://www.dcs.gla.ac.uk/~matthew/papers/WWW7/www98.html

[17] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, "Using Collaborative Filtering to Weave an Information Tapestry," *Commun. ACM*, vol. 35, no. 12, pp. 61–70, 1992.

[18] D. Barreau and B. A. Nardi, "Finding and Reminding: File Organization from the Desktop," *SIGCHI Bull.*, vol. 27, no. 3, pp. 39–43, 1995.

[19] A. T. T. Ying, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[20] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of International Conference in Software Engineering*. Glasgow, Scotland: IEEE, 2004.

[21] K. Schneider, C. Gutwin, R. Penner, and D. Paquette, "Mining a Software Developer's Local Interaction History," in *1st International Workshop on Mining Software Repositories*. Edinburgh, Scotland: MSR 2004, 2004.

[22] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05. New York, NY, USA: ACM Press, 2005, pp. 159–168.

[23] S. K. Card and D. Nation, "Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface," 2002.

[24] R. DeLine, M. Czerwinski, and G. Robertson, "Easing Program Comprehension by Sharing Navigation Data," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 241–248.

[25] M. Robillard and G. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," in *Proceedings of the 24th International Conference on Software Engineering*, Orlando, USA, May 2002, pp. 406–416. [Online]. Available: citeseer.ist.psu.edu/robillard02concern.html

[26] D. Cubranic and G. C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," in *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 408–418.

[27] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins, "Stuff I've Seen: A System for Personal Information Retrieval and Re-Use," in *Proceedings of the 26th annual international ACM SIGIR conference on research and development in informaion retrieval*, SIGIR '03. New York, NY, USA: ACM Press, 2003, pp. 72–79.

[28] K. Utting and N. Yankelovich, "Context and Orientation in Hypermedia Networks," *ACM Transactions on Information Systems*, vol. 7, no. 1, pp. 58–84, 1989.

[29] S. Dumais, E. Cutrell, and H. Chen, "Optimizing Search by Showing Results In Context," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '01. New York, NY, USA: ACM Press, 2001, pp. 277–284.

[30] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu, "Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse," in *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03. New York, NY, USA: ACM Press, 2003, pp. 47–ff.

Appendix A

Cycle Detection Pseudocode

The following pseudocode describes the cycle detection algorithm used by NavTracks to form associations between files.

A.1 Detection of Cycle

```
FOR each Event DO
    IF EventWindow contains Event
        Form Associations [Goto A.2 Forming Associations below]
    END

    // Now add the Event to the EventWindow

    IF EventWindowSize >= MAXIMUM_SIZE
        Remove oldest event from EventWindow
        Add new Event to EventWindow
    ELSE
        // There is room for one more Event
        Add new Event to EventWindow
    END
END
```

A.2 Forming Association

```
//From the last occurrence of Event within the EventWindow to the
//most recent event, form an association for every event in-between:

FIND last occurrence of Event in EventWindow
```

```
IF Event FOUND
    Let Event = Antecedent
    FOR each SubsequentEvent DO
        Let Consequent = SubsequentEvent
        AssociationRepository GET Association(Antecedent, Consequent)
        Association.TIMESTAMP = NOW!
        AssociationRepository SAVE Association
    END
END
```