NavTracks: Supporting Navigation in Software Maintenance

Janice Singer
Institute for Information Technology
National Research Council Canada
janice.singer@nrc-cnrc.gc.ca

Robert Elves
Department of Computer Science
University of Victoria
relves@uvic.ca

Margaret-Anne Storey
Department of Computer Science
University of Victoria
mstorey@uvic.ca

Abstract

In this paper, we present NavTracks, a tool that supports browsing through software. 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 present the reasoning behind NavTracks, its basic algorithm, a case study, and propose some future work.

1. Introduction

Modification and/or enhancement of software require thorough investigation of the program source to determine where to change the code. In turn, investigating software involves navigating through source code and documentation and following different kinds of relationships, such as control flow and inheritance relationships in the code.

Navigation in information spaces has been widely studied in Human Computer Interaction (HCI). Consequently, various tools and/or processes have been proposed to support navigation. These ideas can be transferred to software engineering. That is, a software system can be conceived of as an information space (software space) through which a user has to navigate (cf. [3]). Storey et al. [4] also note that a software space is a hypertext space with many different kinds of hypertextual relationships. In this paper, we explore one HCI-inspired solution to the problem of navigation in software systems, that of tracking interaction histories [5-7].

Sim [3] and Storey [4] identify two types of navigation in software spaces: directed searching and undirected searching (i.e. browsing). Searching occurs when a developer is looking for specific information in the space. Browsing is used to explore the information space and understand high-level concepts. The tool we propose, called NavTracks, is designed to support this type of browsing and the achievement of a high-level conceptual understanding of the code.

According to Sim [3], browsing is most effective when a conceptual organization has been imposed on the data – allowing developers to follow relationships between points in the information space. However, frequently in software spaces the conceptual organization that is imposed does not match the developers' mental models. This is because software spaces are frequently organized according to the hierarchical containment relationships between files, such as class and subclass. However there are possibly other more meaningful file relationships that could legitimately serve as the basis for an organizational scheme supporting navigation. Furthermore, in hierarchical systems, individual files are allocated to only a single location within the hierarchy ignoring other possible classifications that might make just as much semantic sense. Such hierarchical systems also fail to take into account the differences between tasks and individuals that can influence the optimal organization for supporting navigation.

To avoid these problems our tool focuses on the file-to-file relationships established by the developers, as they navigate in the software space. NavTracks presents a view of the related files which reflects the characteristics of the developer's current task as well as his or her individual browsing and file access idiosyncrasies (if any). Related files are determined by examining their navigation path. Following Wexelblat [6], we believe that the path information garnered from navigation in an information space can reveal the user's model of how information should be connected; i.e., the paths can reveal a user's mental model of the system. Consequently, the file relationships and recommendations determined by NavTracks should be consistent with the user's mental model of the code.

NavTracks unobtrusively suggests files that may be of immediate interest to the developer. In essence, our system creates a model of the relationships between files as a developer browses them, and then recommends files that are related to the currently focused-on file. This approach allows a developer to browse a software space by focusing on the relatedness of resources, and not needing to rely primarily on the hierarchical definitions within the file system.

This paper is structured as follows. First, we present a conceptual overview of NavTracks. Second, we look at related research from the software engineering domain. Next, we define essential requirements of the NavTracks tool. Subsequently, we present the architectural and design details of NavTracks. This is followed by some initial evidence on the benefits of NavTracks to support software maintenance. Finally we conclude with a discussion of the limitations of NavTracks and suggest improvements.

2. Conceptual Overview

Imagine you are trying to fix a bug in a part of a program which you have not visited for some time. The fix potentially impacts related files from a cross-cutting concern, e.g. logging user events in a log file. Fixing the bug requires that you understand all of the related code in several files. However, you cannot remember which files those are. There are several tools available for navigation in your IDE, including search tools, bookmarks and cross reference views, however these do not help you to recall the file names.

Fortunately, when you open up one file that you know is relevant to the bug fix, NavTracks displays a short list of recommended files that you immediately recognize as being also related to the logging feature. As you navigate through those files, you see further recommendations of files you should consider and gradually your mental map of the program feature under consideration is reinstated. As you continue to explore, you get confused by which files are important, but the recommendation view helps you keep track.

The advantage of the recommendation view is that rather than forcing the user to recall where the desired file is, it presents a short list of file names thus relying less on the developer's ability to recall and more so on the usually stronger ability to recognize.

The basic premise underlying NavTracks is that navigation patterns reveal relatedness between files. Although the scenario above considers the benefits of these recommendations after a long period of inactivity in a part of the program, we have observed from our empirical work [8] that programmers easily get disoriented when jumping between related files and often interleave programming tasks that involve different sets of files. Hence, such recommendations can be useful even after a short time of browsing and particularly during intense periods of programming multiple tasks. There are several areas of similar research that build on the observation that artifact and process information during development can assist in program comprehension and software navigation. This work is reviewed next.

3. Related Work

Several recent systems mine CVS (concurrent version system) data to provide developers with information about correlated changes between files to facilitate the maintenance process. That is, during development, a recommendation is provided concerning which files to look at when a certain file must be changed. A few different approaches to this problem have been implemented. First, both Ying [9] and Zimmerman [10] independently designed a system that uses CVS data to find files that have frequently been changed together, and then makes recommendations to developers regarding these co-occurrences. Both approaches were moderately successful. In fact, Ying's approach found co-occurrences that would have been difficult to find using simple structural heuristics. Shirabad, et al. [11, 12] also mined CVS repositories to find co-occurrence of changes. In their approach they trained a classifier to determine which relationships between files predicted co-occurrence of change. Then the classifier was used to make recommendations based on its training set. Shirabad et al.'s approach was also able to produce some interesting recommendations.

The primary difference between these approaches and the NavTracks approach of capturing navigation events is that CVS repositories are often out of date when compared to the local version on a developer's workstation – hence they do not provide recommendations based on current browsing patterns. Additionally, while these systems can make recommendations related to software navigation, their primary raison d'être is concerned with impact analysis and source code dependencies. While two files may be dependent upon one another, the dependency may not be direct. NavTracks can reveal these hidden dependencies.

Another area of related research is collaborative software development in the area of awareness. Two of these systems use local interaction history to provide information to developers during a development session. First, TUKAN [13] creates a model of a software system where relationships between artifacts are determined according to structural, temporal, and task characteristics. These relationships are either implicitly or explicitly weighted, creating the possibility of calculating the distance between each artifact and every other artifact. The model is then updated using browsing information of the developer. In a similar system, Schneider et al. [14] create a shadow CVS repository. Without the user's awareness, edits are auto-committed to the shadow repository from the local workspace. The shadow repository is then mined to provide awareness information to developers as they are working. Both of these systems are similar to NavTracks in that they use local information to update the model (i.e., they do not wait for CVS check-ins). These systems differ from NavTracks in that they focus on providing awareness information to the end-user. Awareness information in both systems addresses who touched what parts of code and when, and may or may not be relevant to a particular maintenance task.

Two other systems are worth mentioning in relation to NavTracks as they specifically focus on navigation support. First, Mylar [15] is a degree-of-interest model for the Eclipse environment. In Mylar, each time a program element is selected or edited, its interest value increases, while the interest value of all other elements decreases. Thus at any point in time, the relative values of the program elements reflect the degree to which they have most recently been accessed (i.e., how interesting they are in relation to the current programming task(s)). Unlike NavTracks, Mylar does not consider relationships between elements. This means that when more than one task is currently being worked on, the degree of interest model becomes less relevant in finding related files. However, the Mylar approach is extremely complementary to the NavTracks approach in that both offer information about potentially interesting files based on current activity in the IDE. In fact, where Mylar appears to bring a significant benefit to developers is in the reduction of the number of files displayed in the Package Explorer view of Eclipse. This is a similar goal to that of NavTracks – to allow developers a more efficient navigation pathway through the system.

The second relevant system, FEAT, was developed by Robillard and Murphy [16]. FEAT provides a mechanism for explicitly documenting scattered concerns in the program through the use of a concern graph. The concern graph can be used for navigating to related code in a concern. Unlike NavTracks, however, FEAT requires explicit intervention by the developer to create the concern graphs. However, Robillard and Murphy [17] propose an enhancement to FEAT based on an algorithm for automatically inferring concerns based on navigation pathways. This goal is essentially the same as the NavTracks tool. Their proposal differs in that it uses a stochastic model and is concerned with a different level of granularity, i.e. it is not file based. Automatically identified concerns still require programmer involvement to accept or reject them in the concern graph. The FEAT tool and its proposed enhancement share the same goal as NavTracks which is to improve navigation -- the two approaches could potentially be integrated for optimal support.

In summary, there are a number of approaches that share the goal of assisting in navigation. However, none of them assist in the problem of navigation by implicitly using navigation history to present a short list of relevant files to the file of interest.

4. Requirements

From a review of the literature and our empirical work, we propose the following requirements for the NavTracks tool.

4.1. Non-disruptive

The collection and analysis of developer data should not disrupt the developers work, nor require any additional work. (cf. [18]). Additionally, the collection and analysis of the data should not affect performance. The tool should be readily accessible by the user and integrated into the development environment.

4.2. Current

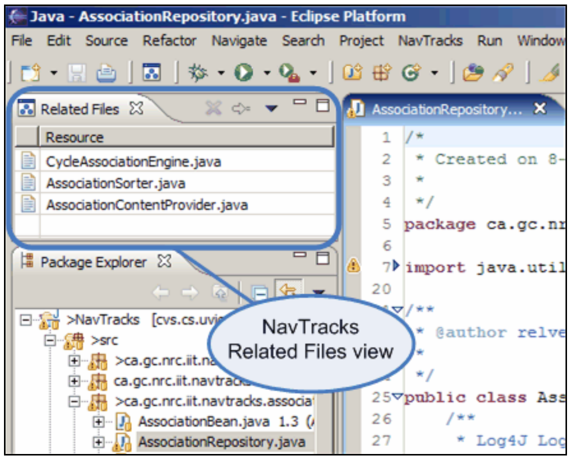
One of the premises behind the NavTracks tool is that recommendations based on current browsing patterns may be useful in the very short term as well as over longer periods of time. Consequently we believe that NavTracks should collect fine-grained events from locally available information. Rather than mining a large (and possibly stale) database of historical traces, NavTracks provides information concerning the recent actions on the local copy of the development project.

4.3. Approximate but Efficient

Many recommendation systems and search engines (e.g. Google) provide results that are not 100% accurate but are very efficient. They are seen as useful and are accepted by most users. Our goal is to ensure NavTracks behaves like these systems in the sense that it is efficient and accurate most of the time. Obviously if its accuracy is below a certain threshold, NavTracks will be rejected, however, we believe that accuracy does not have to be perfect for NavTracks to be accepted. To ensure efficiency, we propose that NavTracks should not be sensitive to network disruption in terms of availability or response time (the limitations of this approach will be discussed in the conclusion).

5. Implementation and Architecture

NavTracks offers developers recommendations for related files given their previous navigation patterns. Figure 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 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.


Java - AssociationRepository.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project NavTracks Run Window
Related Files
Package Explorer
AssociationRepository...
NavTracks Related Files view

Resource
CycleAssociationEngine.java
AssociationSorter.java
AssociationContentProvider.java

```
>NavTracks  [cvs.cs.uvi
  >src
    >ca.gc.nrc.iit.na
    ca.gc.nrc.iit.navtrac
    >ca.gc.nrc.iit.navtracks.associa
      AssociationBean.java 1.3 (
      AssociationRepository.java
```

```
/*
 * Created on 8-
 *
 */
package ca.gc.nr

import java.util

/**
 * @author relve
 *
 */
public class Ass
    /**
     * Log4J Log
```

Figure 1. NavTracks Related Files view

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.

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, it will be discussed later.

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 out.

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.

At the heart of NavTracks is the association engine. Associations are formed using a heuristic based on our observations of navigation patterns of developers - that files that participate in short navigational 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, our algorithm forms associations between the first file in the cycle and each file contained within the cycle. This is illustrated in Figure 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, of minimum size k, within the event window. In our current implementation, we use an event window of size n = 4, and a minimum cycle length of size k = 3 (note this is also the absolute minimum cycle size). 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 2 show how associations are formed. Each letter represents a 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 has been detected, no associations are formed up to this point. Step 4 shows a second event A entering the event window, resulting in detection of a cycle of size 4, 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 3 is detected - B-A-B. The association BA is then constructed.

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 as unique objects with metadata tags indicating the frequency of the association along, with the line last visited, and the time of the last occurrence. If the association already exists, its line number and time stamp metadata are updated, and the frequency metadata is increased by one.

Recommendations are then made based on the data in the association repository. Each time a file is opened or navigated to, the Related Files view queries the repository to get the related files. Before they are displayed, recommendations 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. Currently NavTracks is not aware of Eclipse's refactoring capability. Future work would entail participating in refactoring events so that associations are maintained during file deletion, relocation, or renaming.

When displayed, file recommendations are ranked based on time of occurrence, with more recent relationships appearing higher in the list view. 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.

NavTracks is built on the Eclipse platform. Figure 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.

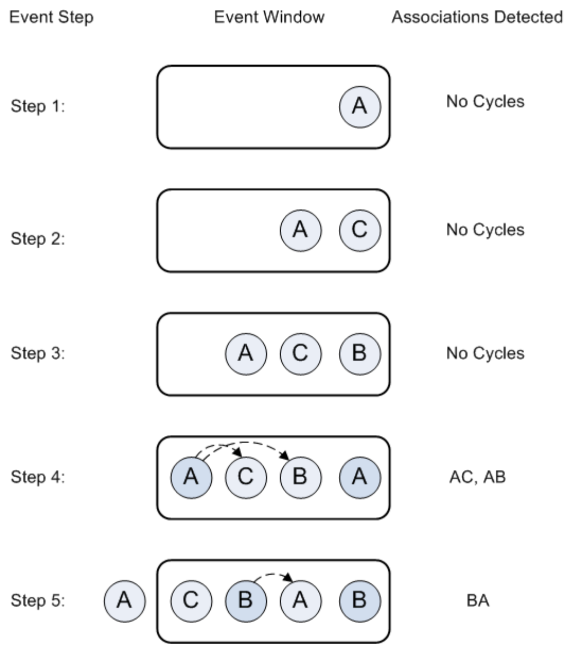


Figure 2. Associations formed via cycle detection, event window size is n = 4, and min. cycle length to be detected is k = 3.

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.

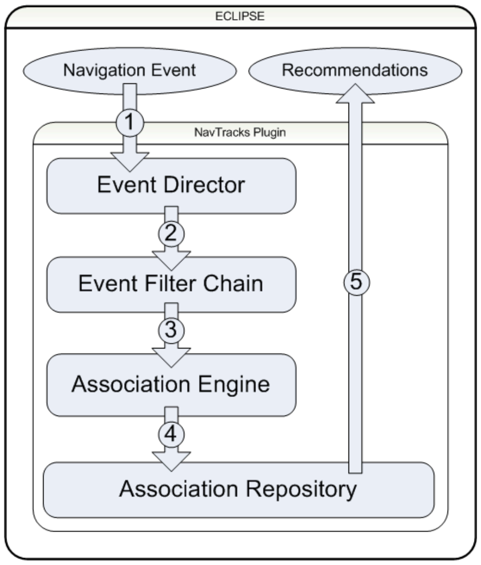


Figure 3. NavTracks architecture.

6. NavTracks Assessment

We assessed NavTracks in two ways. First, we analyzed algorithm performance by collecting the navigation pathways of three developers (Users 1 to 3 profiled in Table 1) to see whether NavTracks would have made correct recommendations for them. Note that the developers did not actually use NavTracks tool.
We simply assessed NavTracks recommendations against their navigation patterns. Second, we conducted a case study of 5 NavTracks users (Users 4 to 8). We asked these users to use NavTracks while they did their everyday software maintenance tasks. We interviewed the users and collected remarks on use, as well as recommendations for enhancements. Table 1 summarizes the user's programming experience, software system and task description.

6.1. Analysis of the algorithm

For the analysis of the algorithm, we captured the event stream (file navigations patterns) of three developers. We placed a data logger on their system to collect all file navigation events. This event stream was used as the basis for assessing the algorithm.

Table 1. User experience description

User ID	Programming Experience	Software System	Task
Algorithm performance analysis users			
1	Professional, 5+ yrs	Java App	Development / Maintenance
2	Co–operative Education Developer	Java App	Maintenance
3	CS Graduate Student	Small Java Apps	Development
Users observed			
4	Graduate Student	Small Java Apps	Development
5	Professional, 5+ yrs	Small Java Apps	Development
6	Co–operative Education Developer	JSP Web App	Major Refactoring
7	Professional, 5+ yrs	Perl Web App	Development Maintenance
8	Professional, 2+ yrs	Java App	Maintenance/ Understand new system

For each event in the event stream, we determined whether NavTracks would have made a correct recommendation for that event. That is, if the NavTracks algorithm would have recommended the next file in the event stream, we counted it as correct, a hit. If the NavTracks algorithm did not make a correct recommendation (the next file in the event stream was not in recommendation list), we counted it as incorrect, a miss. After the event was evaluated, it was used to train the NavTracks algorithm. Thus, we implemented a continuous evaluation and training process on the event stream. Correctness of the algorithm was calculated as the number of hits divided by the number of hits plus misses.

Across the three developers, the average correctness rate of the NavTracks algorithms was 29%, meaning that 29% of the time NavTracks would have made a recommendation that corresponded to the navigation paths of the developers. For each of the developers independently, the correctness rate was 36%, 35%, and 16%. We are not sure why NavTracks performed worse for the third developer.

If we look at correctness according to the number of times an event occurred, we get a slightly different story. We would expect that the more times an event occurs, the more likely NavTracks would be to give a good recommendation – NavTracks has had more time to train itself and hence would provide more useful information. For this analysis, we divided the event stream into event classes based on the number of times the event occurred. Thus, every event in the event stream occurred at least one time, and so belongs in the 1-event class. A smaller number of events occurred 2 times, but all that appeared at least two times (e.g., 2 or greater) belong in the 2 event class. The same goes for the three event class. All events that occurred at least 3 times belong in the three event class, and so on. For each of these classes of events, we calculated the correctness of the NavTracks algorithm. Table 2 below shows this data for each of the developers individually and averaged across the three developers.

Note that as the occurrences in the event stream increase, the accuracy rate of the algorithm also rises (up until about 21 occurrences). Beyond 21 occurrences, the accuracy rate drops slightly. We believe this is because the number of events (file navigations) that occur in this range is much lower, giving us slightly skewed averages.

Table 2: Algorithm accuracy by event occurrence rate

Event Class	D3	D1	D2	Average
2-6 occurrences	25%	23%	17%	**22%**
7-11 occurrences	52%	50%	32%	**45%**
11-16 occurrences	46%	51%	31%	**43%**
12-21 occurrences	60%	31%		**45%**
> 21 occurrences	43%	43%		**43%**

Clearly, there is room for improvement in the performance of the NavTracks algorithm. Nonetheless, our users, as described in the next section, still found NavTracks to be a useful tool.

6.2. User experiences and feedback

The previous assessment looked at the performance of the NavTracks algorithm in terms of its correctness. While this assessment provides us with valuable information, it does not inform us if NavTracks is a useful tool. Hence, we complemented our analysis of the algorithm performance with a case study involving five NavTracks users. Three of the users were expert developers, two of which had at least 5 years experience. One user was a co-op student, the other a graduate student. Three users were maintaining systems consisting of greater than 5k LOC (Lines of Code), while two users were working on systems with less than 5k LOC.

We observed three types of interaction with NavTracks. They are described below. Note that we expect that there will be many more types of interaction. These are just our initial observations.

6.2.1. Newcomer use and New System Development. User 8 was a newcomer to an ongoing development project. This developer was unfamiliar with the code - not knowing where specific code was located within the package hierarchy. Because of this, 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 helped navigation by providing a memory aid for related files. Additionally, the users felt that NavTracks increased productivity by reducing searching.

6.2.2. Wanderer use. User 6 did not have such a successful NavTracks experience. In fact, he reported no valuable recommendations. Through an interview with this developer, we were able to determine 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 will only work well when a developer revisits previously viewed files.

6.2.3. Navigation use. User 7 was involved in a project that required frequent reference to a specific 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 screen when a large number of files are open.

6.3. NavTracks Enhancements

Several of the users suggested enhancements to NavTracks. We have implemented prototypes to match these suggestions.

6.3.1. Maximize Screen Real Estate. Users 5 and 7 found that the Related Files view used too much screen real estate. To address this issue, we are investigating a variety of alternate viewing schemes, including transparency, pop-up windows, and quick views.

6.3.2. Clustering and Visualization. In response to a request by User 7, we implemented a visualization of NavTracks paths as a means of understanding the clustering of related files. We believe that this visualization can aid in recovering the 'implicit' architecture of the system. By implicit architecture, we mean an architecture that is defined by how people move about in a system, as opposed to hierarchical or other relationships that can be used to define system architecture. In our future evaluations, we will assess whether this implicit architecture may aid in program comprehension and general maintenance activities including impact analysis.

Figure 4 shows a visualization of the NavTracks cycles detected during one programming session of an expert programmer. Arcs between files (denoted by boxes) represent an association, with thickness representing the strength of the association (the number of times it was detected). We created the visualization by extending the Creole plugin for Eclipse [19].

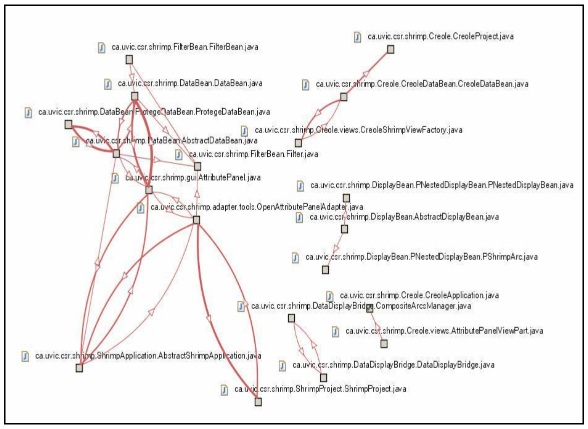

ca.uvic.csr.shrimp.FilterBean.FilterBean.java
ca.uvic.csr.shrimp.Creole.CreoleProject.java
ca.uvic.csr.shrimp.DataBean.DataBean.java
ca.uvic.csr.shrimp.Creole.CreoleDataBean.CreoleDataBean.java
ca.uvic.csr.shrimp.DataBean.ProtegeDataBean.ProtegeDataBean.java
ca.uvic.csr.shrimp.Creole.views.CreoleShrimpViewFactory.java
ca.uvic.csr.shrimp.DataBean.AbstractDataBean.java
ca.uvic.csr.shrimp.FilterBean.Filter.java
ca.uvic.csr.shrimp.gui.AttributePanel.java
ca.uvic.csr.shrimp.DisplayBean.PNestedDisplayBean.PNestedDisplayBean.java
ca.uvic.csr.shrimp.adapter.tools.OpenAttributePanelAdapter.java
ca.uvic.csr.shrimp.DisplayBean.AbstractDisplayBean.java
ca.uvic.csr.shrimp.DisplayBean.PNestedDisplayBean.PShrimpArc.java
ca.uvic.csr.shrimp.Creole.CreoleApplication.java
ca.uvic.csr.shrimp.DataDisplayBridge.CompositeArcsManager.java
ca.uvic.csr.shrimp.Creole.views.AttributePanelViewPart.java
ca.uvic.csr.shrimp.ShrimpApplication.AbstractShrimpApplication.java
ca.uvic.csr.shrimp.DataDisplayBridge.DataDisplayBridge.java
ca.uvic.csr.shrimp.ShrimpProject.ShrimpProject.java

Figure 4. NavTracks Visualization

We are in the process of extending our empirical work. However, this initial assessment has been very valuable. NavTracks is useful in some circumstances, even though the correctness of the algorithm is less than perfect. Next, we explore some of the options available for future work and evaluation.

7. Discussion

In this paper, we presented NavTracks, a lightweight, and non-disruptive tool to support navigation via the recommendation of potentially related files. In this section, we discuss some of NavTracks' current limitations and propose some future work for navigation in software spaces.

7.1. Limitations

We will discuss the limitations of NavTracks in three general areas: granularity, ranking, and locality.

Granularity refers to the level of relatedness of objects in the model. In NavTracks, our atomic unit is the file, augmented by a record of the last line viewed in the file to use as a stand-in for locating a particular method or class definition. Using a method or class definition as the atomic unit may provide more useful information [17]. The benefit of our approach is that it is low cost in terms of the monitoring of the system and is agnostic in terms of the type of textual files associated. Using the last line viewed appears to bring great advantage in placing a historical record for the user to start from (as opposed to placing the developer at the first line when they enter the file). Nevertheless, another level of granularity may be more useful. This requires study. Related to granularity, the relationships that we are interested in are dyadic (relationships between two files). It may be interesting to look at relationships amongst many files. We could do so by enhancing the algorithm to take into account transitive relationships or create one-to-many mappings.

Ranking in NavTracks currently occurs based on recency only. Ranking based on other parameters, such as frequency, may provide better recommendations. The reason we did not use frequency is because frequency is unresponsive to task changes. Initially, we had considered using frequency with a decay function along the lines of that used in the Mylar system. However, it is not clear, how to go about deciding on the appropriate decay function, and whether it should involve the developer. In general, decay depends on activity. In a very active area, decay should probably occur more quickly, whereas for a stable area, decay should probably occur more slowly. In the future we will consider alternative ranking strategies, including a weighted combination of recency and frequency.

A final limitation of the NavTracks approach has to do with locality. Our model is built on the client side. If a developer uses more than one machine, she cannot access recommendations across environments. This could be solved using a client/server model.

7.2. Future work

Our plans for NavTracks center on two areas: continued evaluation and support for collaboration.

7.2.1. Evaluation. Overall, the NavTracks algorithm performs at about 35% recommendation accuracy with an increasing accuracy as occurrences of events increases. Our plan is to correlate accuracy with perceptions of usefulness of NavTracks to determine optimal and minimal accuracy thresholds for NavTracks. However, we received good feedback from our user group with our current level of accuracy.

Our current evaluation is preliminary. Our goal is to continue evaluating NavTracks and collect data and feedback on its benefits and limitations. Currently, NavTracks logs all navigation events and NavTracks usage. We plan to deploy NavTracks with many developers and collect usage data. We will also conduct some smaller qualitative observational studies to assess its use. Both of these methods should also provide us with information about navigation patterns in a software space. To improve the performance of the NavTracks algorithm, we also intend to experiment with different event window and cycle lengths to determine if there is an ideal setting for these parameters for different characteristics of program, programmer and task. Robillard and Murphy [1] in their study of navigation patterns noted that skilled developers tend to have longer navigation cycles. This indicates the need for a dynamic event window in future NavTracks implementations. Additionally, to confirm Robillard and Murphy's results, we will need to conduct more empirical research on navigation to see if we can find patterns in the way that developers investigate source code. Finally, we will evaluate the new interface suggestions received from the existing users.

7.2.2. Collaboration. Support for collaboration centers around the sharing of path information. Both Wexelblat [5] and Chalmers [7] conceive of 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, however, all would require us moving to a client/server model. 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. 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. Software developers tend to stick to one part of the code. By combining tracks, we 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 facilitated only for those already familiar with that information space. It may be that recommendations will not be useful for developers who do not already have a good conceptual model of the software space. This requires study.

8. Conclusions

Investigation of a software space is one of the primary methods that a developer has for understanding, and thereby maintaining, source code. Navigation throughout and within the space is essential to this investigative process. Little research, however, has been conducted on how developers navigate large software spaces and on how to design appropriate tool support for this activity.

NavTracks, through its elegant algorithm and extensible architecture, provides a platform for understanding how tools may be improved for navigation purposes. By experimenting with alternative methods of navigation and comparing diverse tools that assist in navigation, we will gain a better understanding of how more effective tools can be designed to support software navigation in software maintenance.

9. Acknowledgements

We would like to thank the NavTracks users as well as Rob Lintern for his work on the visualizations. Finally we thank Norman Vinson for his valued input.

10. References

[1] M. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," IEEE Transactions on Software Engineering, vol. 30, pp. 889-903, 2004.

[2] L. Moonen., "Exploring software systems," Proceedings of International Conference on Software Maintenance, 2003.

[3] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. M. Cox, "Browsing and Searching Software Architectures," Proceedings of IEEE International Conference on Software Maintenance, Oxford, England, 1999.

[4] M.-A. Storey, F. D. Fracchia, and H. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," Journal of Software Systems, Special issue on Program Comprehension, vol. 44, pp. 171-185, 1999.

[5] A. Wexelblat, "Communities through time: Using history for Social Navigation," in Lecture Notes in Computer Science, vol. 1519, T. Ishida, Ed. Berlin: Spring Verlag, 1998, pp. 281-298.

[6] A. Wexelblat and P. Maes, "Footprints: History-rich tools for information foraging," Proceedings of CHI, Pittsburgh, PA, 1999.

[7] M. Chalmers, K. Rodden, and D. Brodbeck, "The order of things: Activity-centred information access," Proceedings of 7th Intl. Conf. on the World Wide Web (WWW7), Brisbane, AUS, 1998.

[8] M.-A. Storey, K. Wong, and H. Müller, "How do program understanding tools affect how programmers understand programs?" Proceedings of 4th Working Conference on Reverse Engineering (WCRE '97), October 1997.

[9] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," IEEE Transactions on Software Engineering, vol. 30, pp. 574-586, 2004.

[10] T. Zimmermann, P. WeiBgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," Proceedings of International Conference in Software Engineering, Glasgow, Scotland, 2004.

[11] J. S. Shirabad, T. Lethbridge, and S. Matwin, "Mining the maintenance history of a legacy system," Proceedings of Proceedings of the International Conference on Software Maintenance, Amsterdam, the Netherlands, 2003.

[12] J. S. Shirabad, T. Lethbridge, and S. Matwin, "Supporting the maintenance of legacy software with data mining techniques," Proceedings of Proceedings Conference the Centre for Advanced Studies on Collaborative Research, Toronto, Canada, 2000.

[13] T. Schümmer, "Lost and found in software space," Proceedings of The 34th Annual Hawaii International Conference on System Sciences (HICSS '01), Hawaii, 2001.

[14] K. Schneider, C. Gutwin, R. Penner, and D. Paquette, "Mining a software developer's local interaction history," Proceedings of 1st International Workshop on Mining Software Repositories, Edinburgh Scotland, 2004.

[15] M. Kersten and G. Murphy, "Mylar: A degree-of-interest model for IDEs," Proceedings of Aspect Oriented Software Development, Chicago, IL, 2005.

[16] M. Robillard and G. Murphy, "FEAT: A tool for locating, describing, and analyzing concerns in source code," Proceedings of 25th International Conference on Software Engineering, May 2003.

[17] M. Robillard and G. Murphy, "Automatically Inferring Concern Code from Program Investigation Activities," Proceedings of 18th International Conference on Automated Software Engineering, 2003.

[18] P. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving software development management through software project telemetry," IEEE Software, vol. to appear, 2005.

[19] R. Lintern, J. Michaud, M.-A. Storey and X. Wu. "Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse", ACM Symposium on Software Visualization, (Softvis'2003), San Diego, pp. 47-56, and p. 209, June 2003.