

# Easing Program Comprehension by Sharing Navigation Data

Robert DeLine, Mary Czerwinski, and George Robertson  
Microsoft Research, Microsoft Corporation, Redmond, USA

## Abstract

Large software projects often require a programmer to make changes to unfamiliar source code. This paper describes a set of tools, called Team Tracks, designed to ease program comprehension by showing the source code navigation patterns of fellow development team members. One technique shows a list of *Related Items*, given that the user is viewing a given method or class. Another technique shows the *Favorite Classes*, by showing a class hierarchy view that hides less frequently visited classes, methods, and members. Two user studies, a laboratory study and a field study, were run to evaluate the effectiveness of these techniques. The results of the two studies demonstrate that sharing navigation data can improve program comprehension and are subjectively preferred by users.

## Introduction

In large, long-term software projects, team membership and responsibilities change frequently. As a result, a developer must often correct, enhance, or refactor unfamiliar source code. To learn enough about the code to accomplish her task, a developer today typically uses a development environment to read selected parts of the code, to execute the code in the debugger, and to navigate the code through overviews like call graphs, class hierarchies and UML diagrams.

Previously, we described usability problems that we observed during a formative study in which we asked developers to make corrections and add features to an unfamiliar code base [1]. We selected the seven participants, with an average of 18 years of development experience, for their familiarity with the development environment (namely, Visual Studio 2003) and the problem domain (GUI programming). We selected the code base from a software repository web site for its relatively small size (3 KLOC) and its high rating among the site's community. Despite the participants' expertise, we saw several problems as they attempted to complete their tasks:

- They wanted more documentation to help identify the important parts of the code and to describe how the parts are related.
- They got lost as they navigated around the code base, particularly as the number of open documents increased.

- They relied heavily on textual search to find relevant parts of the code, lost time separating good search results from bad, and became distracted by results that seemed relevant but were not.

In this paper, we describe a set of visualizations, called Team Tracks, that addresses these issues and then report the results of two usability studies to evaluate whether these visualizations aid in program comprehension.

## Visualizations based on Shared Navigation

Each of our participants expressed the desire to “pick the brains” of the code's original authors. Such communication, whether in the form of meetings, email or documentation, can have a considerable cost, particularly once the original authors have left the team. To avoid this cost, we instrument the team's standard development environment to record where each developer navigates in the code base as she accomplishes her daily programming tasks. Once per second, our instrumentation peeks at the location of the editor's text cursor and records which project, file, class and member correspond to that location. At this low sampling rate, the user does not notice the recoding, and the storage cost is negligible. We then mine this navigation data to provide visualizations to help those unfamiliar with the code. Our visualizations are 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.

Based on the first intuition, Team Tracks uses the navigation data to filter the typical hierarchical information that the development environment presents about the program. Figure 1(A) shows Visual Studio's Class View, which is a tree that contains the program's projects, the classes in each project, and the members in each class. Figure 1(B) shows our visualization called Class View Favorites, which organizes this same hierarchical information, based on the navigation data.

For each node in the Class View tree, Class View Favorites splits the node's children into those that were frequently visited and those that were not. The infrequently visited items are placed under a new ellipsis node, whose label is based on the item type (“More members,” “More classes,” or “More projects”). By placing the unpopular items under their own tree

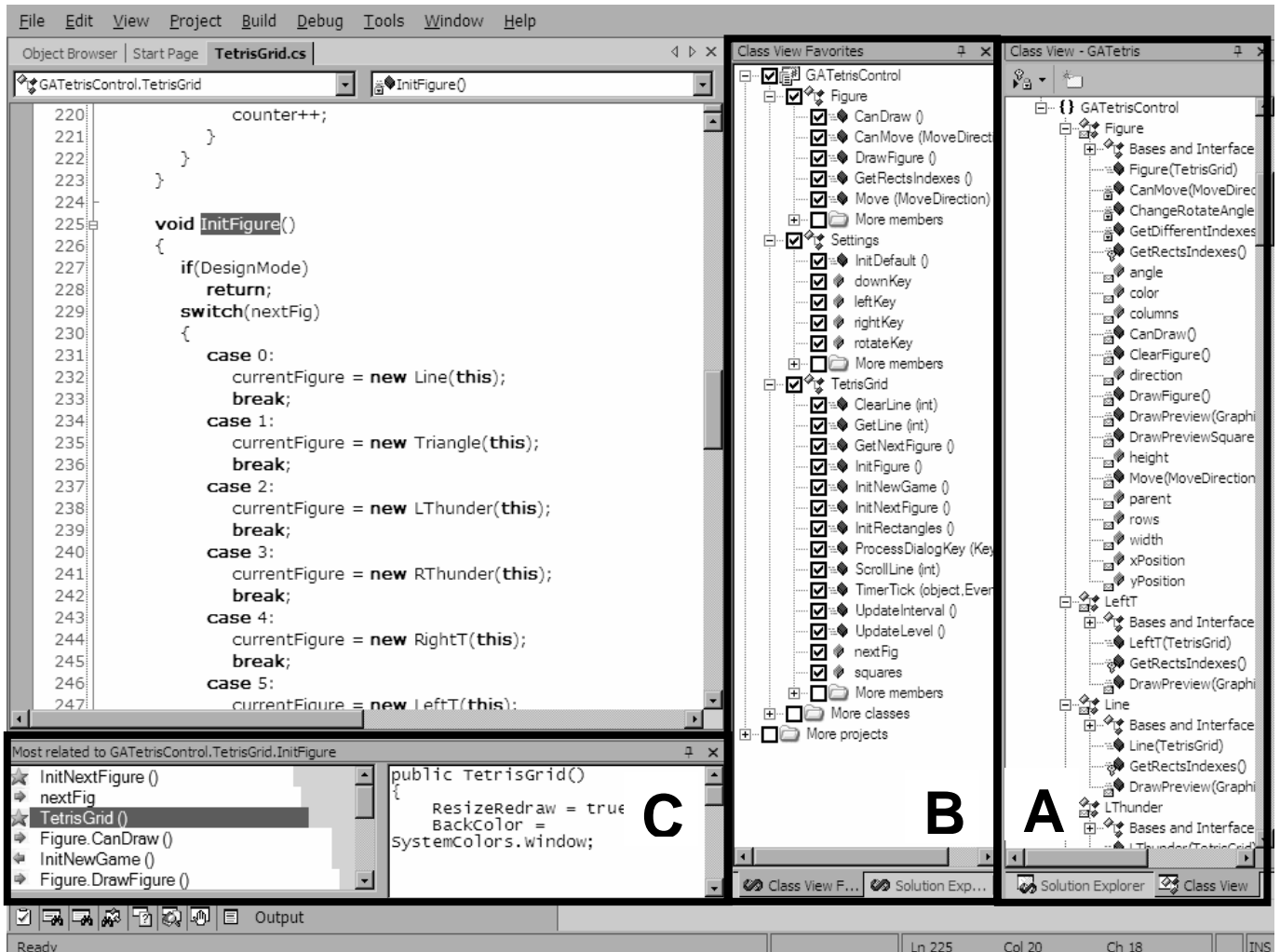





Figure 1. Screen shot with (A) the typical class view, (B) our Class View Favorites, and (C) our Related Items.

node, that node can be collapsed to hide the unpopular items while keeping them accessible. The user can adjust the hierarchy to establish her own working set: unchecking an item moves it under the ellipsis node; checking an item under an ellipsis node moves it up. The projects, classes, and members where the team collectively navigated the most often during development are a small subset of the whole program, as witnessed by the size of the scrollbar in Figure 1(A) and the lack of a scrollbar in (B). Visual Studio also provides a file-based view of the program, called Solution Explorer, and we similarly provide a navigation-filtered version called Solution Explorer Favorites.

Based on the second intuition, Team Tracks provides a visualization, called Related Items, shown in Figure 1(C). Based on the program definition currently selected in the editor (in this case, the method `InitFigure`),

Related Items shows those parts of the program that were most frequently visited either just before or after the selected item. The list is presented in ranked order by frequency of visits, and a bar chart drawn lightly in the background depicts the ranking. In this instance, the team navigated back and forth between `InitFigure` and the method `InitNextFigure` the most often; between `InitFigure` and the field `nextFig`, the next most often; and so on. Selecting an item in the list displays that item's definition in the preview area to the right of the list. This allows the developer to learn more about the item, without losing the current editing context. Double-clicking on an item navigates to that item in the editor.

Because the appearance of an item in the Related Items list is based only on the team's navigation, we cannot provide a reason why the two are related. However, in general, a recommender system is more trusted if it provides a rationale for a recommendation.

Here, we provide an icon for each item in the list as a surrogate for an explicit rationale. A right arrow  means that the definition in the editor directly mentions the list item; a left arrow  means the list item directly mentions the definition in the editor; and a star  is a catch-all that means that no other relationship holds. The fact that `InitFigure` mentions (in this case, calls) `Figure.CanDraw` is not the reason why `Figure.CanDraw` appears in the list, but this fact provides a reasonable guess about why the team navigated frequently between the two. For now, our prototype only looks for two kinds of relationships, namely, A directly references B or B directly references A. We could add other relationships that programmers readily understand, each with its own icon, such as A transitively references B, A and B each reference a third item, code was copied-and-pasted from A to B, and so on. The starred items are the most interesting, and hence we have enlarged the size of these icons to a small degree, since they represent a frequent navigation for which the system could not find one of its known relationships. These we hypothesize to be “diamonds in the rough,” navigationally speaking, for programmers unfamiliar with the code.

## Related Work

Hill, Holland, Wroblewski and McCandless introduced the idea of *computation wear*, namely, capturing a user’s interaction history with a document and making that history available to others interacting with the same document [3]. Their intention was to increase group awareness. More recently, Schneider, Gutwin, Penner and Paquette applied this idea to software artifacts in particular [9]. They store a team’s interactions with a code base in the code’s source control system to increase team awareness, for instance, to show which team member is working on which part of the code. Our contribution is to demonstrate that such interaction histories can be used to help newcomers more quickly learn about shared artifacts.

Team Tracks is most closely related to Kersten and Murphy’s Mylar, an extension to the Eclipse development environment that helps a developer manage her working set [5]. Mylar records an individual developer’s interactions with a code base to compute a degree-of-interest (DOI) function; the more a developer interacts with a program element, the more interest she expresses in it. Mylar then uses that DOI function to highlight related items and elide unrelated items in Eclipse’s user interface. Team Track’s biggest difference is that it persists interaction data and combines the data across all team members. Using data

across team members and across time to filter the user interface makes Team Tracks an instance of social filtering [10].

The Favorites views were inspired by Lee and Bederson’s Favorite Folders [6], which also showed elided material and provided an explicit means for the user to specify which folder items to retain in the view. The difference in Team Tracks, is that the choice of which items to show initially is driven by statistical analysis of a team’s navigation history. Favorite Folders requires the user to manually specify which folders to promote and which to elide.

More generally, there have been many attempts to use software visualization techniques to improve how programmers navigate large and complex programs, in addition to understanding the code better (e.g., [7],[8],[11]). Visualization of code evolution has also been used to help programmers understand why programs are structured the way they are [1] and to assist with software debugging [4]. Many of these techniques use visualization of the actual lines of code, while we have chosen to provide user interface features that provide users with suggestions about what parts of a code base might be more relevant to the programmer’s task.

## User Studies

### Study 1

We brought in 9 software developers who work on small to medium-sized software teams to evaluate the prototype. The average age of the participants is 35 years old (ranging from 29 to 44 years old), and they had been programming an average of 17.9 years. We screened the participants so that they were all familiar with programming in Microsoft’s Visual Studio.net and were experienced in the creation of graphical user interfaces. We ran participants singly in the lab, and an experimenter was present during the sessions.

We ran the study on a Compaq EVO 510 desktop computer with an Intel P4 2.8 GHz processor, 2 GB of RAM and a 40 GB hard drive, running Windows XP Pro SP2. The computer had Dual NEC 18” flat panel monitors, running at 2560x1024 total resolution. We used Visual Studio.net 2003 as the software development platform, with the Team Tracks features added at the beginning of the session. The experimenter initially placed the new features for the user in our preferred locations (for the favorites lists, this was as an alternative tab in each of the Solution Explorer and Class Hierarchy views, and for the related items list this

was at the bottom of Visual Studio, as shown in Figure 1. However, users were allowed to move or resize the windows as desired. Although two users did resize the windows so that they could see more or less of the views, no users moved the windows from their original location.

We used code from the game Tetris for a number of reasons. First, the source code was of a modest, but nontrivial size and complexity. Second, since domain expertise is a critical factor in code comprehension, we chose a game that all our participants had played many times to ensure that they were familiar with the functionality that the program provides. The version we used is called GATetris, available at Code Project at <http://www.codeproject.com/csharp/CsGATetris.asp>.

In order to collect quantitative information about the code exploration process, we developed an add-in to Visual Studio.net to catch various editing, debugging and browsing events. Since text cursor movement is not surfaced as an event, we also poll the cursor at a one-second interval. The logger writes all interaction events with timestamps to a raw data file. A C# customizable script was used to crawl over logs and aggregate the events into statistics and path analyses.

## Tutorial

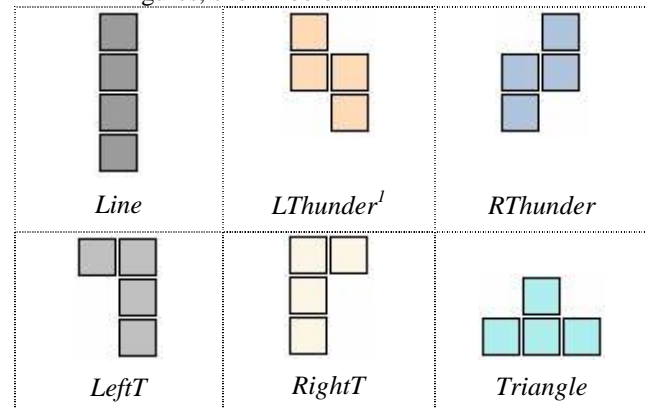
We provided a ten-minute introduction to the new features to users. During this introduction, the experimenter explained that a previous group of developers had been brought in to carry out the exact same tasks that would be performed in the current session and that we had logged their navigation in the code base as they successfully complete the tasks. In other words, we told the participants that the previous developers' well traveled areas of the code provide hints about how to complete the tasks we were going to give them to do. We showed the new Favorites views to the participants, as well as how to interact with the items in those views to jump to different areas of the code. Finally, we explained the Related Items list, its ranking, icons, preview area and navigation affordances. After explaining the new user interface features to each participant and letting him try them out, we began the experimental tasks.

## Experimental Tasks

As in the previous formative study [2], each participant was given an opportunity to explore the source code and program behavior for ten minutes. Then, we asked the participant to do the following four tasks, always in the same sequential order. (We fixed the order because the tasks start out very simple and increase in difficulty and

used the same order as the previous study.) The first two tasks are to answer questions, and the second two are to add features.

1. Which method in the source code determines the next game piece that falls?
2. When a new game begins, the pieces fall at a rate of one grid square per second. What conditions in the game play cause this rate to speed up? Which method contains the logic to increase the speed?
3. The game currently features several different figures, shown below:



However, the game does not feature a square figure like this one:



Add this square figure to the game. Notice that unlike the existing figures, rotating a square figure has no effect – the square looks the same at every rotation angle – which makes it simpler to implement than the other figures. Be sure to update the game logic to ensure that the square figure is a candidate for falling.

4. Change the game so that hitting the space key during game play causes the current figure to fall immediately as far down as it can. This feature spares the player from having to hit the down arrow key many times in succession. The figure's fast fall does not need to be animated. The figure can simply disappear from its

<sup>1</sup> The names of the figures appear throughout the source code. The names *LThunder* and *RThunder* are both inconsistent with the names *LeftT* and *RightT* (L and R rather than *Left* and *Right*) and should presumably be named *Lighting* or *Bolt* rather than *Thunder*. We left the names as-is since inconsistent and confusing names are typical.

current position and reappear at the bottom of the grid.

(Note that the actual game does include the Square figure, which we removed from the source code to support task 3.)

### Dependent measures

We were interested in the users' traversal paths through the code, in addition to their ability to complete the four tasks and solve the quiz questions. We had the quiz responses from the previous group that did not have Team Tracks turned on, so we could compare task completion rates and quiz accuracy between the two groups of participants for the same tasks. We were also curious about whether the users would use the new user interface features, so access to those functions were logged as well. Finally, we collected satisfaction ratings for the new user interface features and importance ratings on a variety of methods and classes from the code, using a subjective rating scale from 1 to 5, with 5 indicating the item is very important for understanding the code. Users' ratings of the importance of the various methods and classes of the program significantly correlated with the previous study participants' tracks through the data, and hence the Team Tracks rankings of importance for those areas of the code,  $r=0.45$ ,  $p=.02$ .

### Results

#### Task completion

Participants were able to find the answers to both task 1 and task 2 100% of the time, which is no different from the previous participants. However, participants using Team Tracks were able to complete tasks 3 and 4 more frequently than the participants without these features. In the previous study, only 1 of 7 participants completed task 3 (to add the square figure) and similarly 1 of 7 completed task 4 (to add the spacebar key functionality). In the current study, 3 of 9 participants completed task 3, and 7 of 9 completed task 4.

One possible explanation for the stronger improvement on task 4 than task 3 is that task 3 involves time-consuming steps that Team Tracks does not address. A typical approach to task 3 involves such subtasks as creating a new source file, copying and pasting code to the new file, applying necessary renamings to the copy/pasted code and getting the new file to compile. These steps are not about learning unfamiliar code and are therefore not improved by Team Tracks.

#### Quiz responses

We compared responses to a quiz on code comprehension between the two groups of participants as

well. The participants from the current study, having the benefit of Team Tracks, did score significantly higher on the comprehension quiz,  $t(16)=-2.04$ ,  $p<.03$ , one-tailed. Team Tracks participants answered quiz responses correctly over twice as often as those who did not perform the tasks with Team Tracks.

#### Feature use

We compared the participants' use of various features with and without Team Tracks present. Developers with Team Tracks turned on were, on average, somewhat less likely to navigate using Find in Files, Goto Definition, or Goto References. In particular, the average number of uses of these features, without and with Team Tracks, were: 2 (without) versus 0.8 (with), for Find in Files; 12 (without) versus 8 (with), for Goto Definition; and 3 (without) versus 2 (with) for Goto References. However, due to our small sample sizes for both studies, none of these differences reached significance at the  $p=.05$  level.

In addition, we logged how often participants used the novel features of Team Tracks. On average, participants asked for a preview of a Related Item 50.1 times; they navigated by double-clicking a Related Items list 20.2 times; they jumped to an item from Solution Explorer Favorites 5.9 times; and they jumped to an area of the code from the Class View Favorites 14.8 times. Solution Explorer Favorites would probably have been used more frequently, but the right-click menu functionality had not yet been completed and was needed for some of the tasks. We feel these numbers reflect that the tool was useful for the tasks, or participants probably would have abandoned using the features in the interest of time as the tasks became more difficult.

#### User satisfaction ratings

The developers appeared to like this first iteration of Team Tracks based on their comments, and their satisfaction ratings generally reflected this, as can be seen in Table 1. The biggest concern was the fact that the developers needed to remember to navigate via the new user interface features and views—something that they were not used to. This was reflected in the “mental demand” question and its average rating, as corroborated by users' closing comments.

Question	Average Rating
Ease of Use: The prototype additions to VS were easy to use (1=Disagree; 5=Agree)	4
Learnability: The prototype additions to VS were easy to learn (1=Disagree; 5=Agree)	4.5

Mental Demand: How much mental activity was required (for example, thinking, deciding, calculating, remembering, looking, searching, etc.)? (1=High; 5=Low)	2.8
Frustration level: How discouraged, irritated, stressed or annoyed did you feel while completing the tasks? (1=High; 5=Low)	1.7
Preference: How much would you prefer this version of VS over existing techniques for developing software? (1=Not at all; 5=Very much)	4.0
How much did you have to divide your attention between the "related items" list and the code displayed? (1=Highly divided; 5=Not divided)	3.7
Global Navigation: The related items list was useful for rapid, global navigation through the code (1=Disagree; 5=Agree)	3.9
Local Navigation: The related items list was useful for minute, local movement through the code (1=Disagree; 5=Agree)	3.6
How useful did you find the list of related items at the lower left of the screen was overall? (1=Not at all useful; 5=Extremely useful)	3.8
How useful did you find the Favorites view of SourceExplorer? (1=Not at all useful; 5=Extremely useful)	3.5
How useful did you find the Favorites view of the Class Hierarchy? (1=Not at all useful; 5=Extremely useful)	3.7
Satisfaction: How satisfied were you with the prototype additions for accomplishing the tasks? (1=Low; 5=High)	3.8

**Table 1. Average questionnaire ratings for Study 1.**

### *Usability issues and user comments*

As mentioned above, users commented positively about the use of a team's navigation paths to help new team members understand a new code base. The participants also provided several ideas for improving the current implementation. In particular, users asked for an alternative view to the Related Items list, where the entire call reference path for a method could be visualized and traversed. Also, in the Related Items preview window, the users wanted to scroll the view to see code just above and below the previewed code. Three users actually started to edit in the preview window, until they realized the window didn't support their actions. Users also asked for a visualization of the

class hierarchy to quickly get a better understanding of the overall architecture. Finally, users had several concerns related to privacy and the ability to track individual programmers on a team. While the experiment's version of Team Tracks does not the user to limit the view to an individual team member's navigation data, the users brought up several cases where this might be quite useful (e.g., viewing data from the most senior developer or architect on the team or the one who "owns" a given feature). Despite these concerns, the participants stated that the benefits of having the tracking information available provided a large enough value that they thought their teams would benefit from having the new system features. Several of the users who were team managers thought that they would have to spend less time reviewing code with new team members if they had these features during the "ramp-up" period for the new developer.

## **Study 2**

In a related study, we examined developers working in an actual, small software development team. The team consists of five developers, four from the company's research division and one from a product team. The team has spent the past two years developing a compiler and program verifier, constituting 100 KLOC, mostly in the C# programming language. Two team members work on the project full-time, three work part-time, and there are additional short-term contributions from interns and contract developers. Given the large size of the code base and the intermittent participation of several of the members, the team was particularly interested in tools for learning unfamiliar code.

We logged the participants working in the code for three weeks, though these were not the initial three weeks that the code had been under development. Hence, this was not an ideal in situ study, but it was a team that was motivated to help us experiment with these ideas and would tolerate prototype-quality tools. After the three weeks of logging, we asked each team member to add the Team Tracks features to their Visual Studio.net environment. We explained each of the Favorites and Related Items views, placing them initially as we did in Study 1. Though we told the team members they could arrange the views any way they preferred, no team members moved our initial placement of the views.

Next, we asked each team member to pick a problem to work on while we observed, and we asked them to try to use our features to see if the features were helpful. We also asked them to think aloud during their tasks.

Our interaction with the compiler team was not at an ideal time for evaluating our tool. Both during our gathering of the navigation data and during our observations, the team was fixing bugs rather than developing new code. As a result, the navigation data is both sparse and occurs in many unrelated parts of the code. For two of the developers, their chosen tasks were in areas for which we had no data, so they had to pick different tasks for the interview. This wasn't a problem, as they had several open bugs to fix in many different locations. However, their ability to steer their task toward the part of the code for which we have data reflects the fact that the team was very familiar with the code base and therefore not the user profile we are targeting. However, since all but two of the team members work intermittently on this project, we hoped that our tool would still prove useful to them.

## Results

All five developers of the product team thought that Team Tracks has merit for new developers on a software project, but because they were deeply immersed in a real software development effort, they experienced different usability issues than the participants in Study 1. For instance, these team members wanted the ability to scope the Team Tracks data both by team member and by time. Because they are generally aware of each others' tasks and responsibilities, this would allow them to narrow the data to a particular feature or bug. They frequently saw that particular team members who were working on specific, unique problems often skewed the data; that is, they were able to sense the immaturity of the navigation data, even in the code areas with the most data. A few of them also mentioned wanting to be able to promote or demote different events in the log, so that methods or classes would not get promoted for the wrong reasons (e.g., frequent bug fixes in an unimportant area of the code).

The icons in the Related Items view were not immediately intuitive, but the arrows seemed to make sense. The enlarged size of the star icon did not necessarily provide value to the developers, and a few mentioned that they would never have figured out why it was a different size if the experimenter hadn't mentioned it. Nor was the star shape necessarily immediately understandable, but it was easy to learn.

A few of the users mentioned that the Related Items view led them to discover places in the code that they wouldn't have thought to look if they hadn't been presented in the list. These opportunities turned out to be especially fruitful as they coded, with one developer

in particular mentioning that he solved his task more efficiently and with higher confidence because he could quickly see how his change affects related (but hard to find) parts of the code. In addition, it was clear that all of the views helped the users resolve any namespace confusions they might have had. (For instance, their code base contains pairs of files with the same name, but for each pair, only one of the files is frequently visited.) Finally, the tool eased their navigation. The code preview was quite often used to determine whether navigating to a related part of the code would be fruitful. In addition, users mentioned that it was useful to be able to navigate directly to related items without having to remember and find them again.

The biggest usability issue we observed was related to remembering to look at the related items window and ensuring that its recommended methods remain visible on screen. This was especially true for the "starred" items in the list that neither referenced the currently selected method nor were referenced by it. It was observed that these instances in the list were truly where the programmers found places in the code that they might not otherwise have known to look. Since this was a team that was fairly senior and had worked on the code for some time together, we take heart that this could be especially valuable to a new project team member unfamiliar with the code.

## Discussion

Both user studies, one in the laboratory and one run with an actual development team performing real tasks, demonstrated that the concepts behind Team Tracks have potential to guide developers' attention and aid navigation to areas of the code that are relevant to their coding goals. In study 1, Team Tracks coders were significantly better at a code comprehension quiz than were participants from a previous study without Team Tracks. In addition, those users rated Team Tracks as easy to learn and use, as well as rating it preferable to Visual Studio without Team Tracks, on average. In the second study, developers who were deeply immersed in an ongoing development project used Team Tracks for the first time to carry out their daily tasks. For those developers that were working in heavily trafficked areas of the code, the related items list helped them remember the name space and eased cognitive load by suggesting potentially important areas to check before coding. Many suggestions for improving the current design were also collected from both studies. We intend to implement several of these ideas, in addition to re-visiting the participants from study 2 after they have used Team Tracks for a few weeks and after their summer interns arrive.



## Future work

Based particularly on the compiler team's feedback, there are several features we will be adding to Team Tracks. First, to alleviate the problem of immature navigation data, we intend to bootstrap the data from the team's source history. We will drive the Favorites visualizations by counting the number of check-ins in which the files and definitions change. To drive the Related Items visualization, we will follow Zimmermann, Weissgerber, Diehl and Zeller [12] and look for patterns in which files and definitions are consistently changed in the same check-in. We also intend to measure the utility of navigation data versus source history data.

Based on the team's suggestion, we have already updated Team Tracks to allow the user to scope the data by person by selecting a particular team member's name or "everyone" from a pull-down menu. We will next add a similar feature to scope the data by time. We will be trying three different units of time: ordinary calendar time; the time between source history check-ins; and the time between the introduction and resolution of a bug in the bug database associated with the code.

Finally, by observing a development team in situ, we see that learning unfamiliar code is only half the battle for a new team member; learning the team's work practices is the other. For instance, for a newcomer to make her first contribution means not only making a change to the code, but also knowing how to build the code into a working system, how to test the code change in the built system (e.g. by executing the system), and how to validate the change before check-in (e.g. running a regression test suite or running static analysis tools). These work practices often change during a product's lifetime and are typically less well documented than the code itself. To help a newcomer learn such work practices, we intend to log all interactions on the desktop and to mine for patterns related to known relevant events, like check-ins.

## References

- [1] Collberg, C., Kobourov, S., Nagra, J., Pitts, J. and Wampler, K. "A System for Graph-Based Visualization of the Evolution of Software," ACM Symposium on Software Visualization, 2003.
- [2] DeLine, R. Khella, A., Czerwinski, M., and Roberson, G., "Towards understanding programs through wear-based filtering," *Proc. Symp. on Software Visualization*, 2005.
- [3] Hill, W.C., Hollan, J. D., Wroblewski, D., and McCandless, T., "Edit Wear and Read Wear," in *Proc. of SIGCHI*, 1992.
- [4] Jacobs, T and Musial, B. "Interactive Visual Debugging with UML," ACM Symposium on Software Visualization, 2003.
- [5] Kersten, M., and Murphy, G., "Mylar: A degree-of-interest model for IDEs," *Proc. Of Aspect-Oriented Software Development*, 2005.
- [6] Lee, B. and Bederson, B. "Favorite Folders: a configurable, scalable file browser," demo paper in UIST 2003 posters and demos booklet.
- [7] Lintern, R., Michaud, J., Storey, M.A., and Wu, X. "Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse," ACM Symposium on Software Visualization, 2003.
- [8] Reiss, S.P. "Visualizing Java in Action," ACM Symposium on Software Visualization, 2003.
- [9] Schneider, K.A., Gutwin, C., Penner, R. and Paquette, D. "Mining a Software Developer's Local Interaction History," 1<sup>st</sup> International Workshop on Mining Software Repositories, 2004.
- [10] Shardanand, U. and Maes, P., "Social information filtering: algorithms for automating 'word of mouth'," in *Proceedings of CHI'95*, 1995, pp. 210-217.
- [11] Wang, Q., Wang, W., Brown, R., Driesen, K., Dufour, B., Hendren, L. and Verbrugge, C. "Evolve: An Open Extensible Software Visualization Framework," ACM Symposium on Software Visualization, 2003.
- [12] T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proc. ICSE*, 2004.

**Author address:** Robert DeLine, Microsoft Corporation, One Microsoft Way, Redmond, WA, USA 98052, Email: rdeline@microsoft.com.