Towards Understanding Programs through Wear-based Filtering

Robert DeLine[1], Amir Khella[2], Mary Czerwinski[1], George Robertson[1]

[1] Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
1-425-705-4972
{rdeline, marycz, ggr} @microsoft.com

[2] Computer Science Department
Human Computer Interaction Lab
University Of Maryland
College park, MD 20742, USA
1-301-405-2725
akhella@cs.umd.edu

ABSTRACT

Large software projects often require a programmer to make changes to unfamiliar source code. This paper presents the results of a formative observational study of seven professional programmers who use a conventional development environment to update an unfamiliar implementation of a commonly known video game. We describe several usability problems they experience, including keeping oriented in the program's source text, maintaining the number and layout of open text documents and relying heavily on textual search for navigation. To reduce the cost of transferring knowledge about the program among developers, we propose the idea of wear-based filtering, a combination of computational wear and social filtering. The development environment collects interaction information, as with computational wear, and uses that information to direct the attention of subsequent users, as with social filtering. We present sketches of new visualizations that use wear-based filtering and demonstrate the feasibility of our approach with data drawn from our study.

Keywords

Computational wear, collaborative filtering, implicit query, program comprehension, software visualization

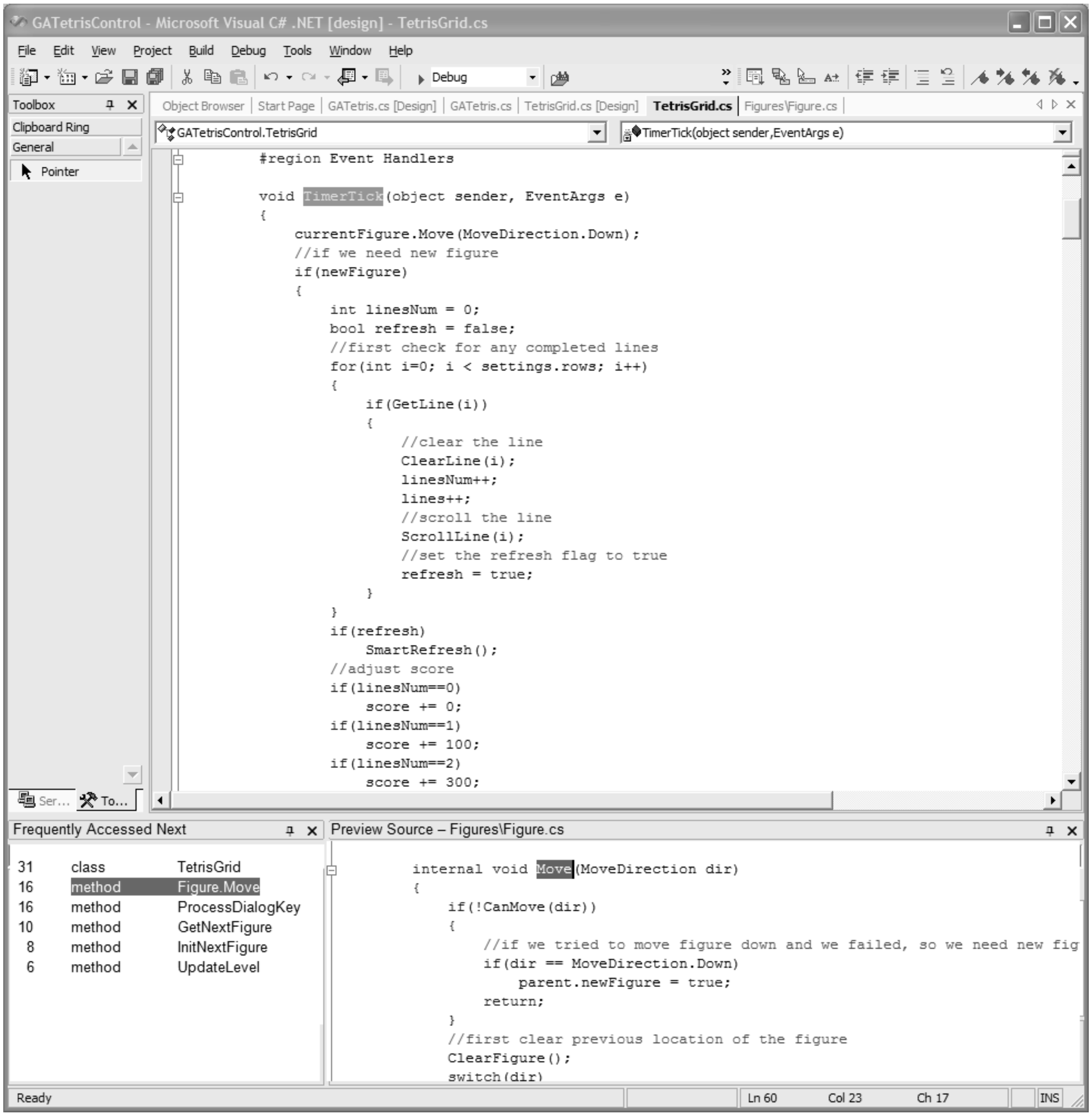
1. INTRODUCTION

In a large, long-term software project, a team programmer must often get to know an unfamiliar portion of the source code in order to fix a bug, to add a feature or to refactor the code to meet a design constraint. The code may be unfamiliar, for instance, because a different programmer was previously responsible for that portion of the code or because the software is in a maintenance phase where responsibility for the code is no longer strictly apportioned among the team's programmers. A programmer facing such a task often relies on little more than the executable code itself. Documentation about the code -- either in the form of inline comments or separate design documents -- is expensive to produce and maintain and is hence often missing or out-of-date. Although researchers have proposed tools to help programmers change unfamiliar code [31], a programmer today typically uses a development environment, like Emacs, Visual Studio or Eclipse, both to learn about the unfamiliar code and to perform the development task. In this paper, we describe a formative observation study of professional programmers using a typical development environment to make changes to unfamiliar code and the usability problems they experience.

To combat the expense of software documentation, we also explore cheap ways to transfer knowledge from programmers familiar with the code to those unfamiliar with it. In previous research, 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 [11]. Other researchers have developed the idea of *collaborative filtering*, namely, allowing users to rate or annotate portions of a large shared information source (e.g. newsgroups) to help one another quickly find high-quality subsets of the information [9][12]. In this paper, we combine those ideas into wear-based filtering. We propose saving programmers' interactions with a project's source code and using that interaction history to direct the attention of subsequent programmers to the important portions of the code.

For instance, Figure 1 shows a sketch of a proposed development environment in which the user's current position in the source code forms an *implicit query* [6] into the code's interaction history. The list at the bottom left labeled "Frequently Accessed Next" shows the definitions (methods and fields) that previous programmers visited most often directly after visiting the current method. In this case, after visiting method TimerTick, developers next visited TetrisGrid 31% of the time, Move 16% of the time, and so on. One of those methods (Figure.Move) is selected and shown in the source preview panel on the lower right.

In order to explore whether or not program code visitation frequency and implicit importance are even correlated with each other, and to discuss these ideas further with real developers, we ran a formative user study on program comprehension. Formative studies are used early in the design of a prototype concept, in order to initially explore the idea with stakeholders in the design and to get user feedback for iterating or even initially designing a prototype. In Section 2, we describe our study and the usability issues that it uncovered. In Section 3, we describe three conceptual designs that address these usability issues and use the idea of wear-based filtering. In Section 4, we compare our work to existing research and conclude in Section 5.


GATetrisControl - Microsoft Visual C# .NET [design] - TetrisGrid.cs
File Edit View Project Build Debug Tools Window Help
Debug
Toolbox
Clipboard Ring
General
Pointer
Object Browser | Start Page | GATetris.cs [Design] | GATetris.cs | TetrisGrid.cs [Design] | TetrisGrid.cs | Figures\Figure.cs
GATetrisControl.TetrisGrid
TimerTick(object sender,EventArgs e)
Ser...
To...
Frequently Accessed Next
Preview Source – Figures\Figure.cs
Ready
Ln 60
Col 23
Ch 17
INS

```
        #region Event Handlers

        void TimerTick(object sender, EventArgs e)
        {
            currentFigure.Move(MoveDirection.Down);
            //if we need new figure
            if(newFigure)
            {
                int linesNum = 0;
                bool refresh = false;
                //first check for any completed lines
                for(int i=0; i < settings.rows; i++)
                {
                    if(GetLine(i))
                    {
                        //clear the line
                        ClearLine(i);
                        linesNum++;
                        lines++;
                        //scroll the line
                        ScrollLine(i);
                        //set the refresh flag to true
                        refresh = true;
                    }
                }
                if(refresh)
                    SmartRefresh();
                //adjust score
                if(linesNum==0)
                    score += 0;
                if(linesNum==1)
                    score += 100;
                if(linesNum==2)
                    score += 300;
```

31	class	TetrisGrid
16	method	Figure.Move
16	method	ProcessDialogKey
10	method	GetNextFigure
8	method	InitNextFigure
6	method	UpdateLevel

```
        internal void Move(MoveDirection dir)
        {
            if(!CanMove(dir))
            {
                //if we tried to move figure down and we failed, so we need new fig
                if(dir == MoveDirection.Down)
                    parent.newFigure = true;
                return;
            }
            //first clear previous location of the figure
            ClearFigure();
            switch(dir)
```

Figure 1. Sketch of an editor using wear-based filtering

2. THE STUDY

2.1 Participants

Seven experienced programmers (all male), average age of 36 years old (range=30 to 42), participated in this study for receipt of 2 copies of Microsoft software gratuity. The programmers were chosen to match a series of profile questions that were used in the screening process. These questions included the requirements that they work on small software development teams, often need to debug or modify the code of other developers, and that they have worked on long-term projects in the past. In addition, we asked the participants about their history using a computer (average=21.6 years), length of time spent programming (average=17.7 years), what their editor of choice was: 5 used Visual Studio.net, 1 used Visual Studio 2003 and Notepad, and another only used Notepad. As for their preferred debugging tools, all reported using the tools that come with Visual Studio, though 1 used FoxPro and CodeWarrior for Palm in addition to

Visual Studio. On average, the participants worked in software development teams of around 13 members.

2.2 Methods and Procedure

Each participant was run singly in our user study laboratory. The experimenter was in the room with the developer the entire time, as this was a "think aloud" study, where the participant tells the experimenter what they are thinking as they work on the code. When participants first arrived in the lab, they were given instructions as to the goals of the study, told that there were no "right or wrong" answers but that we were primarily interested in how programmers approach and comprehend code written by someone else. They were told that they would be given a certain time period (20 minutes) for each task, and that they might not finish, but that this was ok since it was their thought processes that we were interested in. The study was run on a late model Compaq Evo machine with double flat panel LCD monitors running at 2560 x 1024 total resolution. A late model Microsoft keyboard and IntelliMouse were used for input. Windows XP and Visual Studio.Net were used to review the code during the study.

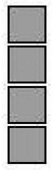
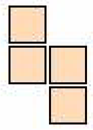
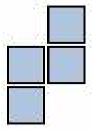
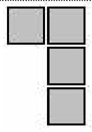
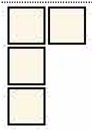
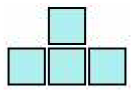
Code from the game Tetris was used in the study for a number of reasons. First, the source code was of a nontrivial size and complexity. Second, since domain expertise is a critical factor in code comprehension, we chose a game that all our participants had played many times to ensure that they were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

In order to collect quantitative information about the code exploration process for each participant, a custom-designed logger was run to track code traversal information. The logger was developed as an add-in to Visual Studio.net to catch various editing, debugging and browsing events. The logger writes all interaction events with timestamps to a raw data file. A C# customizable script is developed to crawl over logs and aggregate the events into statistics and path analyses. In addition, Camtasia Studio version 2.0 [3] was used to videotape the desktop behaviors of each participant.

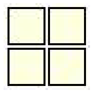
2.3 Tasks

First, each participant was given an opportunity to explore the source code and program behavior for up to ten minutes. Then, the participant was asked to do the following four tasks, always in the same sequential order. (This is because the tasks start out very simple and increase in difficulty.)

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:

 *Line*	 *LThunder*[1]	 *RThunder*
 *LeftT*	 *RightT*	 *Triangle*

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



Square

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 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.)

2.4 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. Finally, we collected 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.

2.5 Results

2.5.1 *Task completion*

Participants were able to find the answers to task 1 100% of the time, and reviewed the code enough to answer task 2 100% of the time. Since they were given a deadline of only 20 minutes to

[1] 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.

solve each task, tasks 3 and 4, which involved actually modifying the code, had lower success rates. Task 3 and 4 were only completed by 1 out of 7 users (a different user each time) in the 20 minutes given, but several users got very close and were headed in the right direction.

2.5.2 *Quiz responses.*

After trying all 4 tasks, participants were asked to close Visual Studio and perform a quiz on their comprehension of the code base. Participants fared well on questions about TetrisGrid and NewFigure (both related to tasks 1 and 2), but had more trouble with other questions related to drawing and animating game pieces (tasks 3 and 4). In other words, participants were able to answer questions for tasks they completed better than for the more difficult tasks that they were less likely to have finished in time. This result is not surprising.

2.5.3 *Correlations between method and class importance ratings and frequency of access*

In addition to taking a quiz on the code base, participants were asked to rate the importance of various methods and classes for the tasks they had performed, in addition to understanding the code overall. A Pearson product moment correlation of those ratings was carried out against the actual frequencies of visiting those areas of the code, across all the participants. The correlation was significant, $r=.79$, $p<.01$. In general, therefore, areas of the code most frequently visited were also rated as more important by the participants.

2.5.4 *Observations*

All participants started by exploring the control flow of the program by looking for places in the project which might be good candidates for an entry point. Participants who started exploring the member variables made faster progress than those who started by looking at functions. Despite the fact that all subjects were familiar with the game's concept, only two of the subjects used top down comprehension to hypothesize that the major functionality should be implemented through a timer. This assumption helped them to locate the main timer event handler using "Find in Files". The remaining subjects used bottom up comprehension and navigated a larger percentage of the code before locating the same function. The two most widely used features for forward navigation were "'Go to Definition"' and "'Find in Files"'.

2.5.5 *Usability issue*

Although our goal was not to evaluate the usability of Visual Studio, we did observe many issues that detracted from an optimal user experience while participants carried out their tasks. Primarily, issues related to navigating and "re-finding" areas of the code that had already been visited detracted from developers quickly accomplishing their tasks. Search ("Find") was also problematic, in that there were several versions (scoped or unscoped) and users often found themselves searching in a limited space when they thought they were performing a global search. In addition, while the tabbed main window was useful for keeping multiple areas of the code viewable within one click, there was a lot of hunting and pecking observed when users forgot what an area of the code was called, or when too many tabs were opened. At some point, most users ended up closing all the tabbed windows and starting over to make the tabs more navigable and to remove clutter on the screen. Some users complained that, though they like the class view, it resized every time they edited the code. It was clear that users thought of the code via its spatial layout, and wanted their class views to maintain a consistent look as well. There was also a lot of effort involved in window layouts across the two monitors. Users expressed annoyance when the layouts were not maintained or optimized for their task at hand. One user had issues with Intellisense not behaving correctly while typing with a compilation error. One user wanted a "clear all" button for when you start something new in the code, and asked for a way to save snapshots of where he was in the code for different tasks in a single project. He said he does not know of many programs that do that and it would really help. There were a few issues related to F-key functionality (e.g., users using F6 when they wanted the functionality related to F5).

2.5.6 *User comments*

Users provided several helpful suggestions and ideas for improving the experience of comprehending code written by someone else. Some users suggested that graphical depictions would be much more easily understood than textual ones for initial code exploration. Visualization that described the high level architecture and abstracts away the detail was mentioned as key. Also, understanding the relationships between components was suggested as helpful. Finally, having easy-to-use tools to help a developer navigate around the code base (e.g. find which code calls a method, or find all instances of a variable, etc.) would improve efficiency and understanding, according to our participants. It was also mentioned that brief, one-line explanations (written by a developer) on certain lines within the code for important processes would make for quick understanding of code areas easier. One participant suggested that the developer should be able to tag code as they explore it, including 1) the main summaries of user input source (keyboard, timer) 2) A UML-type diagram for reviewing the main classes. This user wanted to place these tools on a third monitor, and click on them to get an instantaneous breakdown of the main parts - in other words visually navigate the classes if desired, and 3) Show a drop down for the whole solution A) Namespace B) Class C) Field selector type (data, methods, properties, private) D) Function Selector that would let developers navigate to any part of the program, regardless of what file they are in. Now we support the current file, but in practice, this user said he navigates around the program, and he only wants one file open at a time. In fact this user mentioned that he only usually wanted one function at a time, with a quick way to get back to the functions he was just working on. Finally, some users stated that they would have understood the code better initially if there had been description headers at the top of each code file that explained the contents. Essentially when attempting to trace a method's origin, the developer would use search to find where it existed and how it was used in other places. Any shortcuts for this navigation task were described as useful.

2.6 Discussion

Based on the post-experiment questionnaires, all subjects agreed that finding the entry point and understanding the control flow was the most difficult task since the code was giving them a broader working set than what they need for the task. The large number of members and variables in each class made it difficult for most of them to navigate using class view. Many participants expressed their need to eliminate from view those areas of code irrelevant to the current task, in order to identify areas that have not been explored yet. Moreover, they also expressed the need for an advanced highlighting and annotation tool that they can use with the same simplicity of noting something on paper, while being linked to the code. When asked about what they would say to subsequent code owners, all participants agreed about communicating the names of the important areas in the code. Some participants also communicated a "navigation path" that would lead others through the best route to understand the code.

3. EXPLORING CONCEPTUAL DESIGNS

Throughout the study, participants were not shy about complaining about the lack of inline comments and overview documentation. They wanted to “pick the brain” of the original author, chiefly to get a summary of the game’s major components and their functions and to learn how control flows among these components. The cost of such knowledge transfer, either in the form of documentation or direct communication, can be considerable. Hence, we propose designs that use interaction history as a free source of information about the program.

Here we present mockups of three conceptual designs that use interaction history to address two major problems that the participants experienced: (1) needing to scan much of the source code to find the system’s most important pieces, and (2) getting lost while exploring the code.

3.1 The FAN List: implicit queries of history

In the study, we repeatedly saw the following frustrating scenario. The user is studying a given method and would have questions about other parts of the code, like “What does this called method do?”, “Who calls this method?”, or “How is this referenced data structure represented?” To discover the answers, the user would navigate from away the current method to a part of the code likely to provide answers. In the best case, the user would navigate directly with the “go to definition” command and would return directly afterward. However, in many cases, the navigation would involve textual search, followed by pruning out the search results by visiting each result site. In some cases, the user would navigate several hops away from the original method of interest. The result is that the user would often get lost and have a hard time returning to the original method of interest.

To address this, we combine computational wear, social filtering and implicit queries in our mockup of the Frequently Accessed Next (FAN) List design, shown in Figure 1. At the top of the figure is the usual code editor. At the bottom left is the FAN List. Based on the current definition (method or field) under the cursor in the editor window, the FAN List displays all definitions to which that previous users navigated directly after leaving the current definition more than 5% of the time. Clicking on an item in the FAN List causes the definition to appear in the preview window to the right of the FAN List. This allows the user to inspect the given definition (and the code around it) without changing the current focus in the editor window. (Double-clicking on an item in the FAN List puts that definition in the editor window, so the user can also navigate with the FAN List.)

This design addresses both the problems mentioned above. When the user has a question about the method in the current focus, rather than searching the entire source code, the FAN List displays likely places for the answer to lie because they are the

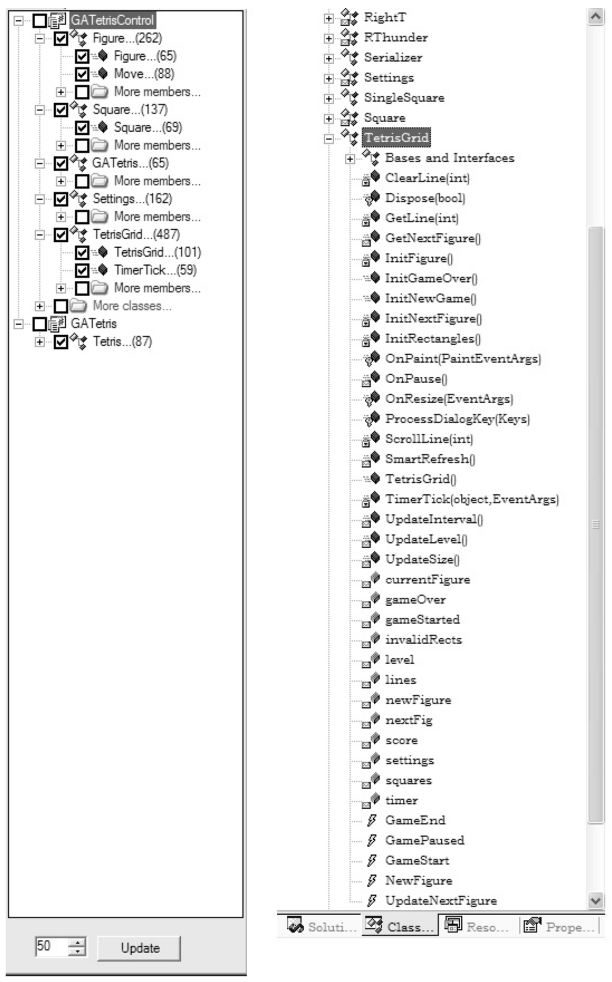


Figure 2. Code Favorites (L) vs Studio's Class View (R)

most related places in the code, according to previous users’ navigation steps. Of course, if the user’s question is sufficiently unrelated to previous users’ tasks involving the focus method, then the FAN List may not point to the most relevant code. The intent is to give the user assistance most of the time.

Second, the preview window allows the user to inspect related code without losing the current focus. This supports questions whose answers are one hop away from the code under focus. For multiple hops, the design could be modified to make the preview window a first-class editor with its own FAN List. The worth of this additional complexity depends on the frequency of multiple-hop questions.

In the current design, the FAN list is ordered by frequency, so that the most visited item appears first. This has two potential problems. First, during the early stages of development of a system, this might become self-perpetuating. That is, something that others frequently access is more likely to be pursued, hence reinforce the existing statistics. Second, the frequency order for some items may be different in different contexts. Programmers may find this disconcerting. An alternative design might have a fixed ordering (e.g., alphabetic or in the same order the items are defined in the source), with frequency information provided textually or graphically. Our intention is to try each of these techniques and evaluate the alternatives iteratively with real users.

3.2 Code Favorites: wear-filtered overviews

The results of the study indicate that interaction history can be used to distinguish parts of the code based on their importance. An initial overview that provides developers with code hotspots should accelerate the comprehension process by highlighting the areas which should receive early attention. Additionally, developers need to specify which of these members are relevant to their tasks and constitute a working set for later use.

We propose Code Favorites, a prototype providing a customizable class browser to navigate classes and members similar to favorite folders [13]. Figure 2 shows Code Favorites (left) versus Visual Studio's Class View (right). Class View displays in a tree view all of the projects in the system, the types defined in those projects and the members defined in those types. Code Favorites, instead, filters the tree based on interaction history. In this case, only members that previously received 50 or more visits are displayed as children of their containing class node. The remaining members are displayed as children of an ellipsis node, labeled "More members", that is a child of their containing type. Similarly, only the most frequently accessed projects and types are shown.

As with favorite folders, checkboxes allow the user to move items in and out of the ellipsis folders. Checking an item in an ellipsis folder moves it out to the parent node; unchecking an item moves it to the parent's ellipsis folder. The novelty of this design over the original favorite folders is seeding the list of favorites based on the team's interaction history.

The main advantage of this design for program comprehension is that it highlights "hot spots" in the code, directing attention to those places where previous programmers have worked most often. Compared with the "Frequently Accessed Next" list, the change to the development environment's existing user interface is modest. The checkboxes that let the user to move items in and out of the ellipsis folders allow the user to establish working sets of types and methods. But of course we intend to evaluate and iterate each of these designs and compare them to each other.

3.3 Wear for degree-of-interest highlights

Several of the participants complained that there was no overview showing the system components and their relationships, either in the form of documentation or as an online visualization. This lack of overview is likely a big contributor to their getting lost as they navigated around the code. Modern development environments are capable of automatically generating UML class diagrams, such as the one shown in Figure 3. Here we propose supplementing or filtering such diagrams based on interaction history.

The UML diagram in Figure 3 shows the various classes in one of the Tetris projects and two kinds of relationships among the classes, namely inheritance ("is-a", shown with triangle arrows) and association ("has-a", shown with open arrows and field labels). Over this diagram we superimpose interaction history data as a heat map with four gradients of red. The field TetrisGrid.-currentFigure and the method TetrisGrid.TimerTick, for instance, were accessed quite often, the field Settings.leftKey less so, and the method SingleSquare.Draw less still. The fields, properties, and methods shown with white background were accessed relatively infrequently or not at all.

The advantage of such a diagram is that it shows the whole system and the degree to which previous programmers worked on various parts. The display and its highlights remain unchanged as the user navigates among the definitions, which should help to keep the user oriented, particularly if a "you are here" marker is kept in sync with the editor's cursor. In addition to the "you are here" marker, we could also highlight the Frequently Accessed Next members. The clear disadvantage is that the diagram is large, detailed, and cannot scale to large systems.

One approach to addressing the scaling problem is to use wear data as a filter, in the style of Code Favorites. We could use ellipses to omit from the lists those fields, properties and methods with white backgrounds. We could also collapse seldom accessed classes, like TetrisGridContainerDesigner, to just their names or omit them altogether. Such a diagram would scale better but would be less useful in keeping the user oriented. The benefits and tradeoffs from each of these designs might change with different task contexts, and hopefully by studying them in situ we can learn general heuristics for optimal presentation based on task type.

3.4 Discussion

3.4.1 Task-specific versus task-neutral data

An assumption behind our conceptual designs is that the frequency of navigation to a definition varies roughly in proportion to the definition's importance. This may not always be the case. For instance, a developer who often fixes obscure bugs will generate interaction data that emphasizes parts of the code that relatively unimportant for understanding the program. Our hope is that aggregating data over many developers doing many tasks will counteract the task-specific nature of each epoch of interaction data.

An alternative approach could take advantage of the task-specific nature of the data. Many development teams encourage the convention that changes to the source code be committed to the source control system in batches dedicated to a single conceptual change, i.e. to a single programmer task. For such teams, the interaction history could be stored and presented per programmer, and per check-in [2]. As a prelude to any of our conceptual designs, we could present the user with a list of check-ins and their textual descriptions. The user could then select a check-in whose description mentions a task similar to the user's current task. Our visualization would then use that check-in's associated interaction history rather than the team's aggregate interaction history.

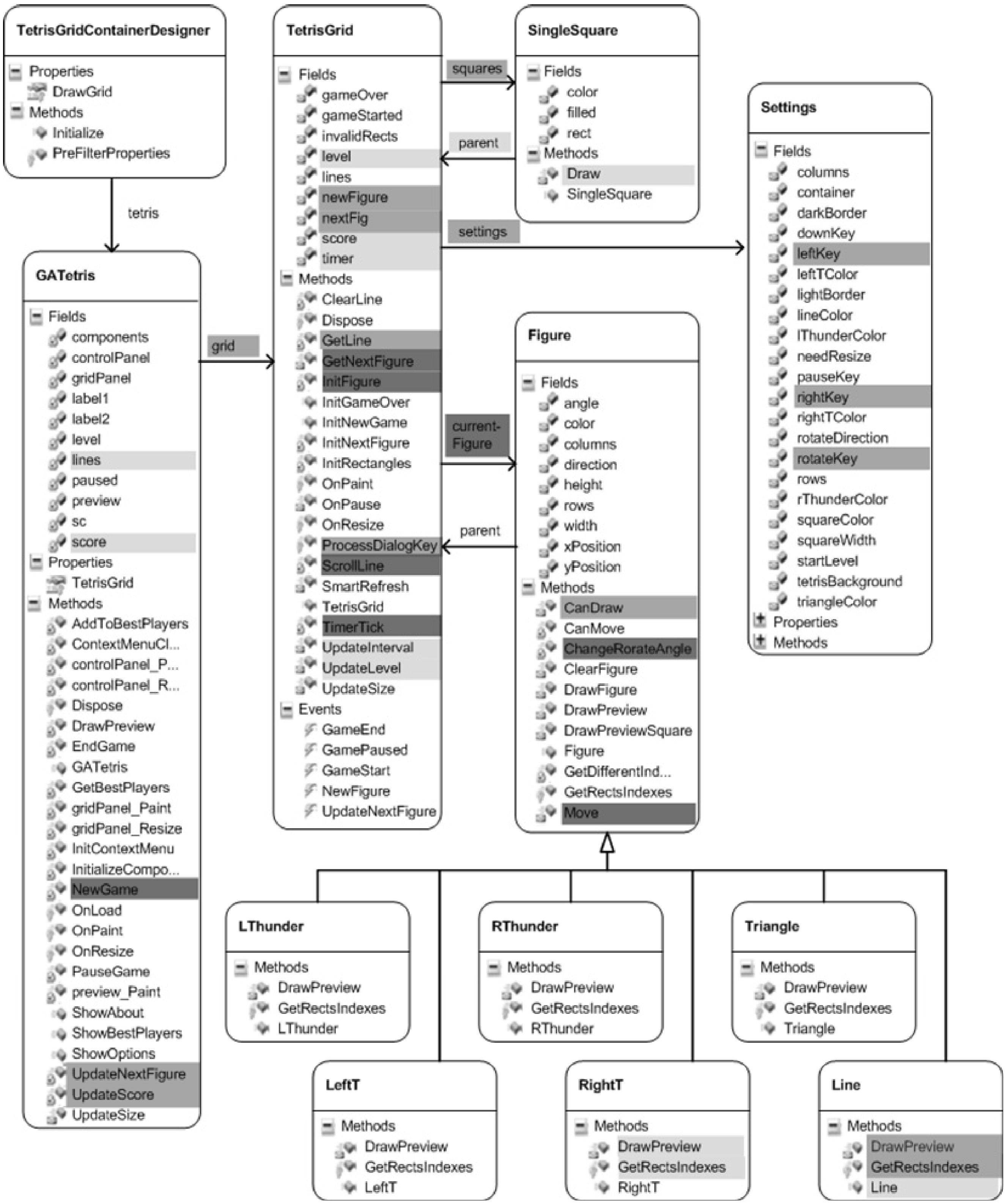


Figure 4. UML diagram with wear heat map

3.4.2 Cleaning up the interaction data

In all of the study sessions, we saw participants make false steps in navigating to a definition in which they were interested. For instance, the participant would misremember the location of a definition or would be distracted by a definition with a similar or misleading name. Recovering from the mistake could sometimes take several navigation steps.

Such mistakes have the effect of tainting the interaction data, which in turn can diminish the quality of our visualizations. To address this, the data could be filtered to weed out these mistakes. For instance, a long visit duration or the occurrence of an edit at the target of a navigation step could be used as a cue that the navigation was successful. Similar care must be taken in bootstrapping the visualizations, which are useful only after a threshold amount of interaction data has been collected.

4. RELATED WORK

4.1 Program comprehension

Previous studies have focused on proposing and validating cognitive models of program comprehension rather than on usability issues with development environments. Here we briefly summarize these previous results.

4.1.1 Cognitive models of program comprehension

Previous work focusing on the psychology of programming identified two main approaches for program comprehension: *bottom up* and *top down* [20]. In bottom up comprehension, lines of code are recognized as functional chunks, which are consolidated into algorithms, and finally into a semantic process. Top down comprehension, also called hypothesized understanding, assumes the existence of an initial clue or hypothesis about the intended functionality of the code. Iterative refinement is used to understand the program in a top down fashion, building more hypotheses at each level and verifying their validity. Additionally, Letovsky [14] introduced the knowledge-based approach, where programmers can employ both bottom up and top down comprehension, depending on the cues that are given to them. Another study by Littman et. al [15] identified two additional, equally important comprehension strategies: *systematic* and *as-needed*. The systematic strategy describes the comprehension process as the use of extensive symbolic execution of the data and control flow between modules to gain a detailed understanding of the program prior to code modification. In contrast, the as-needed strategy tries to minimize the effort required by localizing understanding to only those parts of the program that need to be changed. Finally, Mayrhauser and Vans observed in their study [18] that program understanding is built concurrently at several levels of abstraction, by freely switching between bottom-up, top-down and knowledge-based strategies.

One family of studies has focused on *code beacons,* first introduced by Brooks [2], which are stereotypical segments of code. Wiedenbeck and Scholtz [34] examined the role of beacons in program comprehension. Gellenbeck and Cook [8] showed that beacons include meaningful variable and function names, comments and program structure. Crosby et al [5] investigated the role of expertise in recognizing code beacons. He concluded that "beacons may be in the eye of the beholder," for instance, when more experienced programmers could recognize lines from a binary search algorithm that novices did not distinguish. Beacon identification have been discussed throughout software engineering literature [16][17][32][33] as an essential ability that expert programmers use during program understanding. These findings coincide with Petre's analysis [22] on the ability of expert programmers to organize and use secondary notation in graphical programming to boost readership skills.

4.1.2 Program comprehension studies

Different strategies have been employed to study empirical program comprehension. During a maintenance task requiring comprehension, the measures of completeness, correctness and time required to finish the task [14][15] are frequently used. Recall tests [21][27] are also performed to test subjects' abilities to answer questions regarding a piece of code that they study for a limited time period. Subjective ratings [27] has been used recently to measure different levels of comprehension. Additionally, program comprehension studies may ask subjects to label or group different code members based on the similarity of their functionalities [23]. Soloway and Erlich [30] asked programmers to fill in blank lines and complete unfinished programs on paper in an unfamiliar source code without providing specifications about the program's use or functionality. Similarly, Bertholf et al. [1] asked novice developers to complete incomplete literal programs on paper. Additional techniques to measure program comprehension involved completing incomplete call graphs [19], modifying existing code [28], report a bug [29], or separate source code from two different algorithms. The tasks and measures used in our initial user study were chosen to provide a somewhat different set of metrics, and of course to examine the usefulness of the concept of wear-based filtering.

4.1.3 Graphical versus textual notations

Mixed results have been reported through the literature on the role of text and graphics for program comprehension. While Green and Petre [10] observed that text was faster than graphics for experimental program comprehension tasks, Scanlan [24] reported an improvement using graphical visualizations when comparing textual algorithms and structured flowcharts. Petre [22] attributes the difficulty in understanding program visualizations to the fact that graphical representation have fewer navigational cues, namely secondary notations, when compared to program text: source code implies a serial inspection strategy. Moreover, she observed that experienced readers tend to use parallel textual and graphical information whenever available to assist their comprehension process: They use text as a main source to guide their understanding of graphical representation.

4.2 History-rich Digital Objects

Hill, Hollan, Wroblewski, and McCandless [11] presented the idea of computational wear on digital objects as a similar effect to the wear occurring to physical objects. Computational wear consists of recording the previous activity with digital objects (documents, images, interface elements...) including the events and context that comprise their use. When accessed later, graphical abstraction encoding previous experience is displayed as part of the objects themselves. Edit wear corresponds to authorship changes and read wear corresponds to readership history. Authoring wear has been exploited by Eick, Steffen and Sumner [7] to visualize changes to lines of code for code reviewing purposes.

4.3 Social information filtering

Social information filtering [4] is an automation of the “word-of-mouth” recommendations [26]. In such scenarios, items are recommended based on values assigned by other “trusted” users with similar tastes. Social information filtering systems are becoming widely popular in online shopping, news, blogging, and several other areas. In these domains, users’ navigation and interaction with the system (views, purchases…) are logged and presented later to subsequent visitors. Additionally, users augment this information with their own reviews, ratings and personal recommendations.

4.4 Mining interaction history

Schneider, Gutwin, Penner and Paquette [25] describe a system that logs a programmer’s interaction history and stores that history in the team’s source control system so that it may be shared among team members. Their focus is team awareness and coordination (e.g. which team member is currently editing which code) rather on program comprehension. In particular, we believe that our proposed use of interaction history for social filtering is a new contribution.

5. CONCLUSION

We have presented the results of a formative, observational study of professional programmers making changes to unfamiliar code. The biggest complaint observed concerned inadequate overview documentation about the system and the biggest usability problem involved getting lost while navigating around the source code. To address both of these, we combine computational wear with social filtering. Specifically, we propose gathering interaction data from the programmers’ development environment and using that data to filter the parts of the program shown to future programmers.

In the paper, we presented three conceptual visualizations using wear-based filtering. We are currently implementing prototypes of these visualizations for evaluation with professional programmers. We are also working with a development team to gather a few months of actual interaction data. This will allow us to compare data from real development tasks to the data we gathered in our study and to evaluate our designs based on real data.

6. References

[1] C. F. Bertholf and J. Scholtz, "Program Comprehension of Literate Programs by Novice Programmers.," Empirical Studies of Programmers: Fifth Workshop., Norwood, NJ, 1993.

[2] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies 18*, vol. 18, pp. 543-554, 1983.

[3] "Camtasia Studio," TechSmith.

[4] Cohen, J., editor, "Special Issue on Information Filtering," in *Communications of the ACM*, vol. 35, 1992.

[5] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "The Roles Beacons Play in Comprehension for Novice and Expert Programmers," in *Proceedings of PPIG*, 2002.

[6] M. Czerwinski, S. Dumais, G. Robertson, S. Dziadosz, S. Tiernan, and M. v. Dantzich, "Visualizing implicit queries for information management and retrieval," in *Proceedings of CHI'99*, 1999, 560-567.

[7] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner, "Seesoft-A Tool for Visualizing Line Oriented Software Statistics," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 957-968, 1992.

[8] E.M.Gellenbeck and C. R. Cook, "An Investigation of Procedure and Variable Names as Beacons during Program Comprehension," Empirical Studies of Programmers, fourth Workshop, ed. J. Koenemann-Belliveau, T.G. Moher and S.P. Robertson, Ablex, Norwood NJ, 1991.

[9] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, "Using collaborative filtering to weave an information tapestry," *Communications of the ACM*, vol. 35, pp. 61-70, 1992.

[10] T. R. G. Green and M. Petre, "When Visual Programs are Harder to Read than Textual Programs," in *Human-Computer Interaction: Tasks and Organisation, Proceedings {ECCE}-6 (6th European Conference Cognitive Ergonomics)*, 1992.

[11] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless, "Edit Wear and Read Wear," in *Proceedings of CHI'92*, 1992.

[12] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl, "GroupLens: applying collaborative filtering to Usenet news," *Communications of the ACM*, vol. 40, pp. 77-87, 1997.

[13] B. Lee and B. Bederson, "Favorite Folders: A Configurable, Scalable File Browser," UMD 2003.

[14] S. Letovsky, "Cognitive processes in program comprehension," first workshop on empirical studies of programmers on Empirical studies of programmers, 1986.

[15] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," first workshop on empirical studies of programmers on Empirical studies of programmers, 1986.

[16] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, pp. 44-55, 1995.

[17] A. Von Mayrhauser and A. M. Vans, "Identification of dynamic comprehension processes during large scale maintenance.," *IEEE Transactions on Software Engineering*, vol. 22, pp. 424-437, 1996.

[18] A. V. Mayrhauser and A. M. Vans, *Program Comprehension During Software Maintenance and Evolution*: IEEE Computer Society Press, 2001.

[19] P. W. Oman and C. R. Cook, "Typographic Style is More than Cosmetic," *Communications of the ACM*, vol. 33, pp. 506-520, 1990.

[20] N. Pennington, "Comprehension strategies in programming," Empirical Studies on Programmers - Second workshop, Norwood, NJ, 1987.

[21] N. Pennington, "Stimulus Structures and Mental Representations In Expert Comprehension of Computer Programs," *Cognitive Psychology*, vol. 19, pp. 295-341, 1987.

[22] M. Petre, "Why looking isn't always seeing: readership skills and graphical programming," *Commuunications of the. ACM*, vol. 38, pp. 33--44, 1995.

[23] R. S. Rist, "Plans in programming: Definition,Demonstration, and Development," Empirical Studies of Programmers, 1st Workshop, 1986.

[24] D. A. Scanlan, "Structured flowcharts outperform pseudocode: An experimental comparison," *IEEE Trans. Softw. Eng.*, 1989.

[25] Schneider, K.A., Gutwin, C., Penner, R. and Paquette, D. "Mining a Software Developer's Local Interaction History," 1st International Workshop on Mining Software Repositories, 2004.

[26] U. Shardanand and P. Maes, "Social information filtering: algorithms for automating "word of mouth"," in *Proceedings of the CHI'95*, 1995, pp. 210-217.

[27] B. Shneiderman, "Measuring computer program quality and comprehension," *International Journal of Man-Machine Studies*, vol. 9, pp. 465-478, 1977.

[28] B. Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Communications of the ACM*, vol. 20, pp. 373-381, 1977.

[29] B. Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Communications of the ACM*, vol. 20, pp. 373-381, 1977.

[30] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," Readings in artificial intelligence and software engineering, 1986.

[31] M.-A. D. Storey, K. Wong, and H. A. Muller, "Rigi: A Visualization Environment for Reverse Engineering," 19th International Conference on Software Engineering, 1997.

[32] S. Wiedenbeck, "Beacons in computer program comprehension.," *International Journal of Man-Machine Studies*, vol. 25, pp. 697-709, 1986.

[33] S. Wiedenbeck, "Novice/Expert Differences in Programming Skills," *International Journal of Man-Machine Studies*, vol. 23, pp. 383-390, 1985.

[34] S. Wiedenbeck and J. Scholtz, "Beacons: A knowledge structure in program comprehension," in *Designing and Using Human-Computer Interfaces and Knowledge Based Systems.*, G. Salvendy and M. J. Smith, Eds. Amsterdam, The Netherlands: Elsevier, 1989, pp. 82-87.