A Field Study on Fostering Structural Navigation with Prodet

Vinay Augustine*, Patrick Francis*, Xiao Qu*, David Shepherd*, Will Snipes*, Christoph Bräunlich†, Thomas Fritz‡

*ABB Corporate Research
Raleigh, NC, USA
{vinay.augustine,patrick.francis,xiao.qu,
david.shepherd,will.snipes}@us.abb.com

†Wotan Engineering GmbH
Otelfingen, Switzerland
braeunlich@wotan.ch

‡Department of Informatics
University of Zurich
Zurich, Switzerland
fritz@ifi.uzh.ch

***Abstract*—Past studies show that developers who navigate code in a structural manner complete tasks faster and more correctly than those whose behavior is more opportunistic. The goal of this work is to move professional developers towards more effective program comprehension and maintenance habits by providing an approach that fosters structural code navigation. To this end, we created a Visual Studio plugin called *Prodet* that integrates an always-on navigable visualization of the most contextually relevant portions of the call graph. We evaluated the effectiveness of our approach by deploying it in a six week field study with professional software developers. The study results show a statistically significant increase in developers' use of structural navigation after installing *Prodet*. The results also show that developers continuously used the filtered and navigable call graph over the three week period in which it was deployed in production. These results indicate the maturity and value of our approach to increase developers' effectiveness in a practical and professional environment.**

***Index Terms*—Software Maintenance, Structural Navigation, Code Recommendation, Field Study**

I. Introduction

Software maintenance tasks, such as fixing bugs or adding features to an existing program, are a common and time-consuming aspect of software development. Studies have shown that effective developers working on maintenance tasks use code search to locate potentially relevant code elements and then use structural navigation techniques to explore these elements and their dependencies [1], [2]. The term *Structural navigation* refers to the traversing of code based on its structural relations, such as method calls or inheritance, rather than simply scrolling or browsing through the file organization. Research has also shown that developers create more functionally correct code when they have structural information in the form of UML diagrams during maintenance tasks [3]. The lesson is simple: developers who actively leverage structural navigation are more effective than those who do not.

In spite of these benefits of structural navigation, many developers instead employ an ad-hoc strategy of browsing for relevant classes or methods and reading any adjacent code. This is partly because modern integrated development environments (IDEs) support unstructured navigation by default, making it easier than structural navigation. Instead of following references, developers spend much of their time scrolling in the editor and opening file after file to find relevant code elements. As a result, they mentally consume much more information than is necessary to complete the task [4], [5].

The goal of this work is to shift developers toward more effective program comprehension and maintenance habits by providing an approach that fosters structural code navigation. In particular, we focus on visualizing call graph information. Previous approaches that extend default IDE capabilities with call graph visualizations predominantly focus on always-on views. These views usually present unfiltered call-graph information, which as a result contains both relevant and irrelevant methods [6], [7].

We build upon this and other previous research work with our approach called *Prodet*. Our approach provides the user with a navigable view of the most contextually relevant portions of the call graph centered on the method currently in focus. Our approach filters the call graph using a live context, which includes information on a developer's recent interaction with the IDE, as well as a set of heuristics on a given element's relevance. This filtering focuses the view on the relevant information to avoid overloading the developer with too much information. The major focus of the presented work is to evaluate whether our approach, based on a combination of recent advances in research, increases structural navigation an industrial setting. Specifically, we pose the following research question:

RQ: Can we foster structural navigation in a developer's work with an approach that provides a filtered and navigable call graph?

To investigate this question, we conducted a longitudinal repeated measures field study with 15 professional software developers. During the six weeks of the field study we tracked the structural navigation behavior of participants in the IDE. Participants worked with only built-in navigation commands as usual for the first three weeks of the study. For the second three weeks, we asked them to use *Prodet*.

The results of our field study show the approach was effective, in that participants used significantly more structural

navigation after installing *Prodet* . Furthermore, the results show that professional developers continuously used the navigable call graph for their work over the period of the study with no major performance issues, indicating the value and maturity of our implemented approach.

This paper makes the following contributions:

- It presents *Prodet*, an approach and Visual Studio plugin that provides a filtered and navigable call graph view. *Prodet* provides a graphical navigation view that uses a developer's recent activity to filter the presented information.
- It presents the results of a longitudinal field study with professional developers to evaluate *Prodet*'s effectiveness at fostering structural navigation.

II. Prodet

To foster developers' use of structural navigation, we developed *Prodet*, an approach that combines recent research advances on navigation and search to improve concern location efficiency. *Prodet* consists of three user-facing features to support developers' work flow and reduce their cognitive load: one feature for navigation, one for search and a map for tracking recently visited code context. The main focus of *Prodet* is the navigation support that provides call graph information and recommendations for relevant methods. The map view shows a graph of recently visited methods, while the search feature is based on our previous work Sando [8]. A screenshot of *Prodet* implemented as a Visual Studio plugin is presented in Figure 1. The novel *Prodet*framework combines efficient program analysis and contextual inference components that allow the approach to instantaneously present relevant context information to the developer without disrupting her work flow. In the following, we will first present a short usage scenario of *Prodet* to demonstrate how it supports a developer's work flow before we will discuss the overall framework, the details of computing relevance of code elements, and the user-interface components.

A. Usage Scenario of Prodet

To illustrate *Prodet*'s features and how it supports a developer‘s work flow, we present a scenario on the use of our approach to fix a bug in a role-playing game. As the developer starts out to investigate a bug on the appearance of monsters in the dungeon, she starts with typing in a search term to locate the bug. As soon as she starts typing, the search view provides auto-complete suggestions. The developer chooses the suggestion "NextLevel" and relevant results are presented in the search view. After skimming the search results, the developer selects the method `GameManager.NextLevel` that appears relevant to her. Once selected, Visual Studio opens the class file of the method opens in and editor and *Prodet*'s navigation view displays the method and its call graph, similar to the one shown in Figure 2. The multiple level call graph allows the developer to quickly determine that the method is relevant, so she starts exploring the callers and callees of the method, such as `AddRandomMonstersByLevel`.

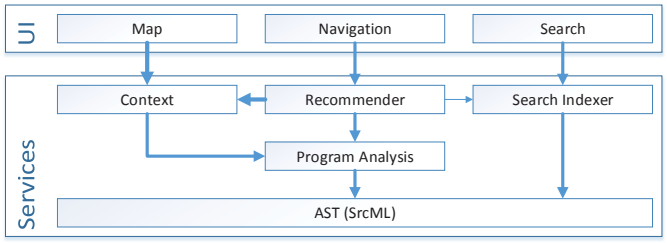

Fig. 3. Prodet Components and Dependencies

Every time she selects another method, it moves to the center of the view and the call graph reflects callers and callees of the method. A decoration scheme colors previously viewed and related methods by class, while leaving related but not viewed methods gray. After further exploration and reading the corresponding source code, our developer is confident that she found the places in the code that need to be changed for fixing the bug. By switching to the map view, she now gets a quick overview of the visited classes, methods and relations.

B. Framework

Prodet's framework is composed of two parts depicted in Figure 3, the user interface components and the underlying services. The Abstract Syntax Tree (AST) service supports all components of our Visual Studio plugin, the search as well as the program analysis that is then used by the recommender component for the navigation support view, and the context component for the map of the recent history.

AST. The AST service generates and updates abstract syntax trees (ASTs) for multiple languages and represents the basis of all data used in the approach. The AST service relies on SrcML [9] to parse C, C++, and C# source code files and generate XML files that capture the AST of the source code. Based on a user-inputted list of source directories and by monitoring the Visual Studio IDE for file save events, the AST service detects all relevant file changes and continuously keeps the ASTs up to date. Testing this service on Prodet's source code with 79 KLOC and a thousand files showed that the processing can be done very fast with 2.1 KLOC processed per second (37 seconds overall) on an engineering laptop with a quad-core Intel Core i7 CPU, 16GB of memory, and an SSD. After initially parsing the entire code base, updates to subsequent file changes, such as when the user saves a file, are essentially instantaneous and not noticeable by the user. The high performance AST service enables the quick responsiveness of our Visual Studio plugin required by professional software developers.

Search Indexer. Our search component, based on our previous work Sando [8], incorporates recent advances in code search research. Sando parses the AST generated by SrcML for C, C++ and C# and classifies program elements as methods, fields, classes, comments and property elements. It then indexes each program element as individual document after performing common preprocessing steps, such as splitting identifiers (e.g., "OpenFile" to "Open" and "File") and stem-

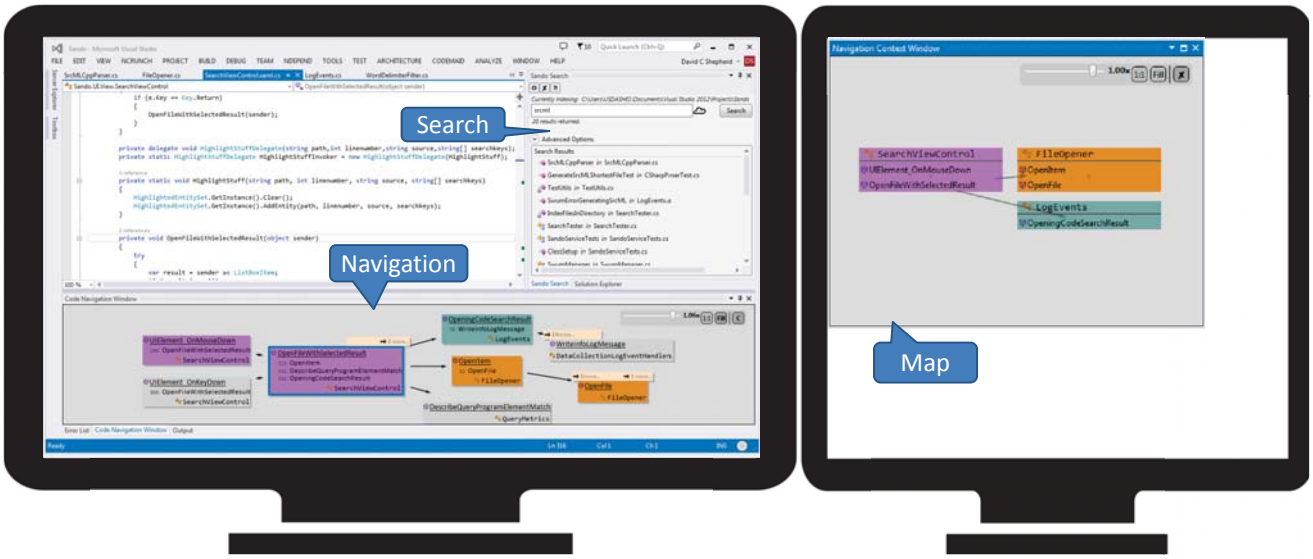

Fig. 1. Screenshot of Recommended Prodet Configuration with Views on Two Monitors

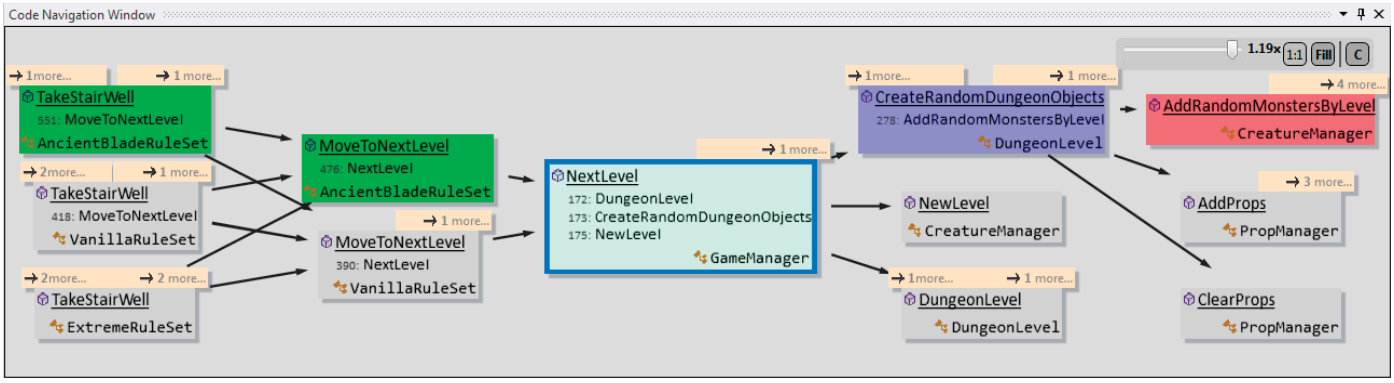

Fig. 2. Navigation View with Call Graph Centered on `GameManager.NextLevel`

ming terms (e.g., "opening" to "open"). This indexing process only requires an additional 22% time on an engineering laptop with respect to the AST-generation time and thus is negligible. Sando uses a vector space model and tf*idf scoring to compare a search query (a set of terms) against all documents and then returns the most related program elements for the query.

Program Analysis. The program analysis component captures two aspects, first, the generation of a global model of the program structure from the source files including call relations, and second, support for query operations on this structure. The program analysis component continuously keeps the model up-to-date using the updates from the AST service. Creating the global structure for the 79 KLOC Prodet source code takes 8 seconds (at a rate of 9.91 KLOC/second) on an engineering laptop. Once the program structure for the entire codebase is generated, the program analysis component responds instantaneously to individual file change events from the AST service.

Once the global model of the program structure is generated, clients can query the model. Usually clients interact with this component by first querying for the method program element that is underneath the current cursor location in Visual Studio, and then clients query for related elements, such as callers and callees of the method or the class it is in. To determine called methods, this component attempts to match the method call to a method definition by testing for the call type, the name and the number of parameters, and then makes a best-effort attempt to determine calling type by performing variable resolution. This approach allows it to resolve 12,000 method calls per second with minimal CPU load. The focus of our analysis is on code the user can access. It does not attempt to resolve method calls in 3rd party libraries, unless stubs, such as header files in C and C++, are included.

Context. The context service is responsible for tracking a developer's *live context* — code elements that are relevant to the current work of the developer. *Live context* in our approach refers to the recent navigation history and recent interactions with the IDE, such as selects and edits of program elements. Similar to the approach by Kersten et al. [10], each time a user interacts with a program element, the relevance score of the element increases. At the same time, to ensure that the current *live context* does not become unreasonably large, each interaction also causes a decay of the relevance score for all

other elements in the *live context*. Thus, the relevance score of elements that are not interacted with will eventually become zero and they will be removed from the current *live context*.

In our performance tests, the context service had no discernible impact on Visual Studio's performance. The most expensive part of maintaining the user's *live context* is mapping the cursor location to the corresponding program element. The latter is done by querying the program analysis service that supports rapid querying of the program structure and is negligible with respect to the user's experience.

Recommender. To provide relevant recommendations to users, *Prodet*'s recommender service determines the relevance of callers and callees of a given method based on a set of heuristics. We apply six different heuristics (described in Section II-C), each of which determines a relevance score for a given method. The overall score, which we refer to as the "degree-of-relevance" (DOR) of a method, is then calculated by normalizing each of the six scores (DOR_1 to DOR_6) with its maximum value to a range between 0 and 1, and adding up the six scores. The output of this service is a list of a method's callers and callers ranked by relevance. The approaches of prior work generate recommendations based on either a single method or a set of seed methods(e.g., [11], [12]). We add the *live context* information as suggested by Shen et al. [13], [14] into the DOR calculation, resulting in better tailoring of the recommendations to the current work of the developer.

C. Live Context and Relevance Heuristics.

Each of the six heuristics in our approach takes into account parts of a user's *live context*.

Query-Based (DOR_1). This heuristic is based on the idea that recent code searches performed by a developer and the search results are related to the task at hand. Our *live context* keeps track of the last search performed together with the search results and the relevance score of each result based on tf*idf. For a given method, this heuristic then returns the relevance score of the method for the last search performed or 0 if the method was not in the result set.

Topology-Based (DOR_2). This heuristic is based on the idea by Robillard et al. [15] that the topology of a method's call graph can help determine its relevance using measures for *specificity* and *reinforcement*. Specificity captures the assumption that elements with fewer structural dependencies are more unique and therefore more relevant, while reinforcement captures the idea that elements that are related to other relevant elements are also more relevant. Intuitively, these measures rank candidate methods highly that are strongly connected to a method of interest and less strongly connected to other methods.

Textual Similarity-Based (DOR_3). This heuristic is based on the assumption that methods are more relevant if they are textually similar to the recent activity of a developer since they are likely to capture related concerns. This heuristic therefore takes a candidate element's name [13] and compares it using *Levenshtein Distance* [16] with the currently selected element from the *live context*. Since a study by Robillard and Ratchford [17] suggests that the similarity of whole element names does not necessarily perform well during recommendation, we first split identifiers and stem terms before comparing them.

Context-Based (DOR_4). This heuristic is based on the observation that users only interact with a limited and small set of files during their work on a task and that recently and frequently visited files are therefore more relevant. This heuristic is calculated similar to the degree-of-interest (DOI) by Kersten et al. [10] that is based on the recency and frequency of interactions with a program element. Our *live context* keeps track of all elements with a positive DOI and updates them for each interaction. The heuristic then just returns the DOI value for a given method if it is in the *live context* or 0 otherwise.

Distance-Based (DOR_5). We created this heuristic based on our assumption that code elements, such as method calls, closer to the current focus of the developer are more relevant than elements further away. This heuristic takes the current cursor position of the developer that is also recorded in the *live context* and calculates the spatial distance of a call to a candidate method from the cursor.

Visited-Based (DOR_6). Studies have shown that developers frequently revisit code [18] and suggest that recently visited elements are the most likely target of a developer's next step [11]. To capture this pattern, our sixth heuristic assigns a score of 1 to the last visited element and 0 to all others. We use this rough approximation and only assign 1 to the last visited element since our exploratory tests have shown that assigning 1 to more than the last element, e.g. the last 10 elements, biases users towards exploring the same paths over and over again and was not beneficial.

D. User Interface (UI)

The UI is composed of three views, one for search, one for navigation and a map view for the recent history.

Search (Sando). The search view is based on our previous work Sando [8], a project-wide code search tool meant as a replacement for built-in tools such as Visual Studio's $FindInFiles$. This type of tool is commonly used to search the currently open project, often during feature location. Unlike other existing tools, Sando is based on recent innovations in applying information retrieval techniques to code search [19], a technique that has been shown to outperform the current state-of-the practice [20]. IR-based search tools provide many advantages over regular-expression-based search tools, such as ranked results, the ability to cleanly handle multi-word queries, and the ability to search for all forms of a given word. These advances considerably reduce effort for developers in both creating a query and reviewing results.

Sando is implemented as a tool window in Visual Studio containing an input box and a list of results (Figure 1, labeled *Search*). Upon typing into the input box, auto-suggestions are provided based on Sando's knowledge of terms and phrases used in the code base [21]. These suggestions, as well as post-

query corrections, further reduce developer effort. Developers no longer have to memorize or have prior knowledge of the project's vocabulary to form an effective query.

Navigation. The navigation UI represents the main component to support our goal of fostering structural navigation. Our tool builds upon prior work that provides an always-on call graph including Blaze [6] and Stacksplorer [7]. The provided view improves upon these existing presentations in two ways. First, our view provides a more balanced visualization, avoiding the depth-only or breadth-only approach of Blaze and Stacksplorer, respectively. Second, our view avoids information overload by applying aggressive filtering, showing only the most relevant portions of the call graph.

An example of the *Prodet* navigation view is presented in Figure 2. It displays the currently selected method in the center with arrows connecting related methods that are recommended by the underlying recommender component. Each method is shown as a box with the method name at the top-left, the class name on the bottom-right, and relevant call-sites (with line numbers) listed below the method name. For instance, the upper-leftmost node in the figure represents method `TakeStairWell` in class `AncientBladeRuleSet` and shows the call-site on line 551 to `MoveToNextLevel`.

In this example, the focus is on method `GameManager.NextLevel`. The navigation view displays callers and callees that are considered relevant by the recommender up to two levels out from the method in focus. Additional connected methods that are not considered relevant enough by the recommender based on their score are hidden and indicated by overlays (e.g., those labeled with "2 more..."). By filtering the call graph using the recommender and only showing the most relevant caller and callee methods based on the six heuristics and the live context, we try to avoid overloading the user with information. In addition, we also enhance the presented call graph with context information: methods currently in the context, i.e. methods that were recently selected, are colored according to their class (e.g., `TakeStairWell` and `MoveToNextLevel` in `AncientBladRuleSet` are both green), whereas all methods not in the *live context* are colored gray. By balancing between depth and breadth of the call-graph displayed in the view, we attempt to provide the developer information that is more helpful to determine their next step.

For a given method the navigation view is constructed using a simple expansion algorithm that executes depth first expansions in both directions until available space is filled. This algorithm supports developers in making next-step decisions without further review of the source code by showing breadth at the first caller and callee level and then opportunistically showing depth.

Finally, the navigation view is linked to the code editor such that when the user selects an element in the view, the corresponding code is opened in the code editor and vice versa.

Map. The map UI shows the navigation context in a separate window (Figure 1, labeled *Map*). It groups all code elements in the context by type (i.e., class) and shows the structural dependencies between each element using arrows. This UI attempts to reduce cognitive load on the developers by memorizing their navigation behavior for them. The assumption is that it might be particularly useful to provide the developers a clear overview of the navigation history and allow them to navigate easily when the window is placed on a separate monitor.

The map UI is interactive. First, the developer can click on each element and be directed to the corresponding definition in the code editor. Second, the developer can also remove undesirable elements from their context by either removing all elements (by clicking the "X" button at the top right of the window) or individual elements (by right-clicking on an element and selecting "remove"). Removing elements from the map causes the context-based DOR to be reduced so that the elements are also de-emphasized in the recommender and the navigation view. This interactivity allows developers to tune the context from a human perspective. It might be useful when they decide that particular elements are not related to the current task or when they are switching tasks.

III. Longitudinal Field Study

To evaluate the effectiveness of *Prodet* in fostering structural navigation we conducted a six week field study with 15 developers. In particular, we investigated the following research questions with our study:

RQ: Can we foster structural navigation in a developer's work with an approach that provides a filtered and navigable call graph?
- (a) Does the amount of structural navigation increase with the installation of *Prodet*?
- (b) Does *Prodet* also foster an increase in a developer's use of Visual Studio's built-in commands for structural navigation?

To address these questions, we monitored each participant's interaction events in Visual Studio using $Blaze$ [22] and analyzed the gathered data.

A. Study Design

We designed the study as a repeated measures study, measuring changes in individual behavior over time. The study was blocked into two three-week periods. The first period, conducted prior to the deployment of *Prodet*, was used to capture a baseline of developers' use of Visual Studio's built-in structural navigation commands. In the second period, participants were asked to install the Prodet tool. This post-deployment period was used to measure changes in their structural navigation behavior.

For this study, we solicited participants among software developers at ABB, an international industrial engineering equipment and power systems manufacturer. We encouraged participation with a raffle for a prize among all participants who used *Prodet* by the end of the study. Overall, we were able to recruit fifteen participants for this study. Of these fifteen

TABLE I
SAMPLE DATA FROM BLAZE'S EVENT LOG

Time	User	Event	Classification
22:04:49	N3	View.Code Navigation Window	Nav.Prodet
22:04:51	N3	View.SourceFile	View
22:04:52	N3	View.OnChangeCaretLine	Nav.Unstructured
22:04:53	N3	View.OnChangeCaretLine	Nav.Unstructured
22:04:58	N3	View.CallHierarchy	Nav.Structured
22:05:00	N3	View.OnChangeCaretLine	Nav.Unstructured
22:05:19	N3	View.SourceFile	View
22:05:22	N3	Edit.Find	Search
22:05:30	N3	Edit.FindNext	Search

participants, seven installed *Prodet* during the second three-week period of the study. The other eight developers did not use *Prodet* and their data was collected to determine whether a change in navigation habits might be due to other effects.

The study kicked-off with a globally broadcasted pre-deployment webinar on *Prodet*, the concepts and benefits of navigating structurally, and information on the study and prize drawing. At the beginning of the fourth week, we held another webinar and encouraged developers to install *Prodet* through email. At the end of the sixth week we closed the study and awarded the prize to a randomly selected participant.

B. Data Collection and Analysis

During the six weeks of the study we monitored participant interactions within the Visual Studio IDE using our monitoring tool $Blaze$ [22]. Developers could install $Blaze$ without installing *Prodet*. Blaze captures nearly all available events that stem from developer actions in Visual Studio. Table I illustrates a sample of the data log that $Blaze$ generates. Each row represents an event with a time-stamp, a unique ID for the developer, the event name recorded from Visual Studio, and an event category.

For our analysis, we wanted to measure developers' level of structural navigation before and after installing *Prodet*. We considered the use of *Prodet* as structural navigation and differentiated between structural navigation events within *Prodet* and the events from Visual Studio's built-in structural navigation commands. In particular, we defined the following metrics to measure structural navigation from the logged data of all developer-initiated events in Visual Studio:

$$N_A = count\ of\ all\ events\ foreach\ developer$$
$$N_P = \frac{count\ of\ Prodet\ navigation\ events}{N_A} \times 1000$$
$$N_S = \frac{count\ of\ structural\ navigation\ events}{N_A} \times 1000$$
$$N_C = N_S + N_P$$

The metric for all interaction events (N_A) counts every event in Visual Studio captured in the log. Events that count towards navigation in *Prodet* (as the numerator of N_P) are any user mouse click on a method or class shown in the *Prodet* tool suite. Events that we categorized as structural navigation in Visual Studio (as the numerator of N_S) are: *Go To Definition* (F12), *Preview Definition* (Alt-F12), *Go To Declaration*, *View Call Hierarchy* (Ctrl+K Ctrl+T), *Find All References*, *Find Symbol*, *Class View* (Ctrl+W, C), *Navigate to Event Handler*, *Navigate To* (Ctrl+,), and the use of the *Object Browser*. N_C then represents the sum of all structured navigation in Visual Studio and in *Prodet*.

To prepare the data for analysis and to provide a balanced set of data on a per-developer basis, we aggregated the data by calculating the average per day for N_S and N_C for each 3-week period. Prior to aggregating, we filtered the data to exclude days with fewer than 100 N_A events logged for each developer because we considered these activity levels as insufficient. Data from the first three days in pre and post sessions were excluded to reduce the bias that we introduced by explicitly suggesting structural navigation as something positive in the webinars. Since developers work at different rates, we normalized the counts by dividing the count of structured navigation events by the total number of events per developer. We then scaled the values by multiplying them by 1000. Finally, only the 15 developers with data in both pre and post deployment time periods were included in the analysis.

C. Results

During the six week period of our study, we logged an average of 36,000 events per developer in the pre-deployment period and 46,000 events per developer in the post period. Participant developers did not work every day in Visual Studio. During the first three weeks, there was an average per developer of 6.5 days with reported data, which rose to an average per developer of 9.5 days with reported data in the three weeks post deployment.

Figure 4 shows the values of N_S in blue (light) and N_P in orange (dark) on a daily basis averaged by developer by day for developers who used *Prodet*. In comparison, Figure 5 presents the distribution of N_S for developers who did not use Prodet over the six week period. While there is a visible increase in structural navigation for *Prodet* users, developers without *Prodet* appear to have a fairly consistent usage of structural navigation throughout the study period, except for the first few days after the first webinar. These graphs already suggest that *Prodet* generally fosters an increase in a developer's structural navigation.

For analysing the difference in structural navigation with and without *Prodet*, we performed a pair-wise comparison of the N_C metric for each developer in the pre and post session. The distribution for the N_C data per developer is presented in Figure 6. The box plots depict the distribution of the data for pre and post deployment in the two groups: the eight developers who did not use *Prodet* (NonUser) and the seven ones who did (ProdetUser).

A Shapiro-Wilk test on the collected data resulted in p-values of 0.45 for pre and 0.80 for post deployment data, indicating that the data is not normally distributed in either period [23]. We therefore use a two-sided Wilcoxon Rank Sum test for paired data in the following that does not assume normally distributed data. We use the paired evaluation of the test because the paired test allows each individual to serve as their own control when the samples are repeated measurements of the same subject before and after treatment [23].

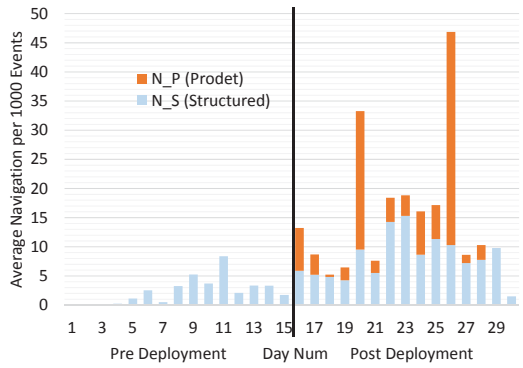

Fig. 4. Average N_P and N_S per User by Day for Prodet Users

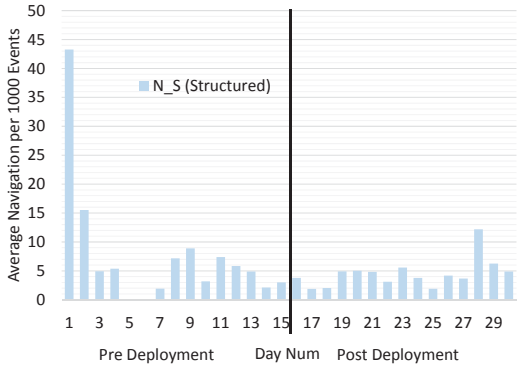

Fig. 5. Average N_S per User by Day for Non Users

Applying the Wilcoxon Rank Sum test to N_C for *Prodet* users pre and post deployment results in a p-value of 0.031. This shows that there is a significant difference (at the significance level of 0.05) in the structural navigation between pre and post deployment of *Prodet* and that developers used significantly more structural navigations once they started using *Prodet*. Comparing the medians in the number of structural navigation events also shows that developers performed more than twice as many structural navigation events after installing *Prodet*.

To determine whether this also holds when we only focus on the use of Visual Studio's built-in commands for structural navigation during the study, we also investigated the difference in the N_S metric pre and post deployment. Figure 7 shows that the median and range of N_S for *Prodet* users increases after they install *Prodet*. A Wilcoxon Rank Sum test results in a p-value of 0.020, indicating a significant difference in the distribution of N_S pre and post *Prodet* deployment for *Prodet* users at the 0.05 level. This shows that the use of *Prodet* did not only increase the structural navigation within the *Prodet* tool itself, it also significantly increased the use of Visual Studio's built-in commands for structural navigation.

Since aspects other than the deployment of *Prodet*, such as listening to the first webinar, could have caused an increase in structural navigation behaviour over the six week period for all developers, we also analyzed the data for NonUsers (developers that did not deploy *Prodet*). A Wilcoxon Rank Sum test produces a p-value of 0.31 showing that there was no significant difference in structural navigation behaviour for the 8 NonUsers between the first three weeks and the second three weeks. The boxplot of the N_S metric presented in Figure 7 also shows a similar picture with the distributions of values for N_S and especially the median being similar. This indicates that NonUsers did not significantly change their use of structural navigation during the study and that the change in structural navigation for the developers who installed our tool resulted from *Prodet*.

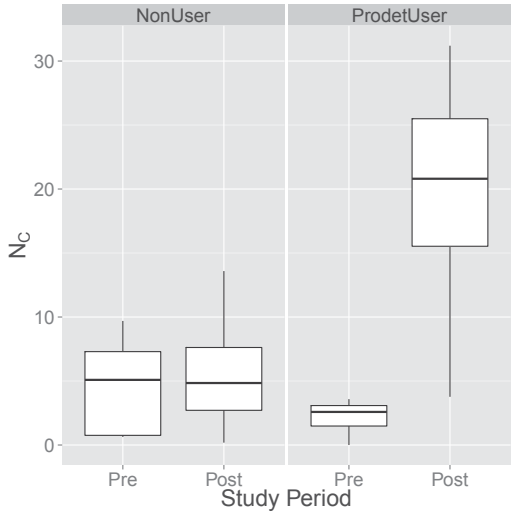

Fig. 6. Distribution of N_C by User Type and Study Period

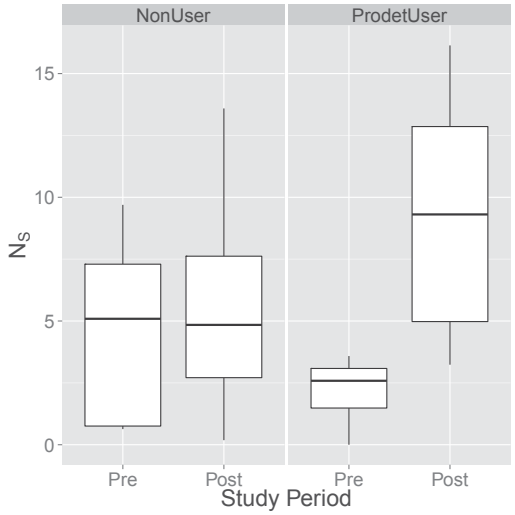

Fig. 7. Distribution of N_S by User Type and Study Period

D. Discussion

Our data shows that developers that installed *Prodet* more than double their amount of structural navigation compared to their original baseline behavior. Furthermore, they also more than double their use of Visual Studio's built-in structural navigation commands after installing *Prodet*. Both increases

were statistically significant, while study participants that did not use *Prodet* had no significant change in their structural navigation.

While there was a small difference in the use of structural navigation commands between ProdetUsers and NonUsers within the first three weeks (before deployment), with NonUsers having slightly more, this difference is not significant and can thus be attributed to random variation in how developers use structural navigation. The spike in structural navigation for NonUsers directly following the introduction to *Prodet* is most likely related to the introduction we provided in which we mentioned the benefit of structural navigation and the "power of suggestion". The overall fairly steady usage of structural navigation for NonUsers during the rest of the study, also supports this assumption. Therefore, for our analysis, we excluded the first three days after the introduction as well as the three days right after the deployment of *Prodet*.

The orange (dark) bars in Figure 4 indicate that developers did not just use *Prodet* right after deploying it, but also kept using it later on. To further investigate the usage of the individual *Prodet* features, we plotted the usage of each in Figure 8. Each bar represents the number of uses of the feature per 1000 events for that day. Overall, the collected data shows that while the search and the navigation were used throughout the second three week period, the Map view (black outline color) was not used enough to appear on the chart. We hypothesize that the lack of usage of the Map view might be due to it being in a separate window and requiring additional screen space. The continuous use of the Navigation view on the other hand suggests that this feature provided value to the user and also that our implementation was efficient and responsive enough for the developers.

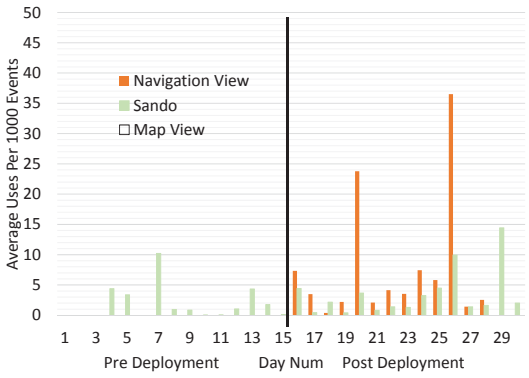

Fig. 8. Usage of Each View

To gather some impressions from developers on *Prodet*, we also collected feedback from participants after the study. Participants generally reported that *Prodet* reduced their navigation effort and helped them understand the code, suggesting that *Prodet* provided value for their work flow:

I am using Prodet tool and finding it extremely useful and found it much simpler and faster than Code Map present in Visual Studio. Thank you for sharing this tool. - One ABB developer.

When I need to go through the new project or new class method calls, I use the code navigation window to see and analyze the method called by different classes. I have suggested [the] Prodet tool to my colleague who was new to C# and the project, to navigate through the classes and methods using code navigation. I personally feel it is helpful in understanding the code using code navigation very much. - Another ABB Developer.

Negative feedback about the *Prodet* tool suite was only given with respect to the amount of memory required to support large codebases ($\geq$ 1MLOC). Developers were typically able to specify a subset of their codebase to reduce the amount of memory used. However, this approach presents a barrier to tool adoption as developers have an additional step to take before starting their tasks.

Overall, the results of our study show that *Prodet* influenced developers and helped them to improve their navigation practices by fostering structural navigation.

E. Threats to Validity

Measuring structural navigation involves the categorization of events into structured and unstructured navigation. Our categorization of events may not be accurate to the intent of the operational definition of structural navigation depending on how the developer uses some commands. We tried to mitigate this risk by discussing our categorization with developers and analyzing several pilot sessions.

The Hawthorne effect, where improvement occurs because we are seeking a change in a particular practice, could impact our results since participants received a presentation on the study objectives prior to the start of this study. This may have raised participants' baseline N_S metric during the pre deployment period, resulting in a lower observed change in structural navigation. The Hawthorne effect may also have influenced participants' behavior since they were incentivized to use *Prodet*. This may have had an influence on the increase in their structural navigation with the *Prodet* tool. We tried to mitigate this risk by separately analyzing the participants' structural navigation using Visual Studio's built-in commands.

Participants in the study were not randomly selected; they self-selected to participate and install *Prodet*. Thus, *Prodet* users may have been more willing to adopt new techniques and try them than a general developer population would be. If this is true, we could see less improvement as the tools move from early adopters to the majority population. Further studies are needed to investigate this aspect.

IV. Related Work

A. Structural navigation tools and recommendation systems

Though navigation features are available in popular IDEs, such as Eclipse and Visual Studio, they have many well-known shortcomings [6]. Most importantly, these navigation tools introduce two key friction points into structural navigation:

information overload and activation effort. Typical call hierarchy views (i.e., call graph views) require programmers to work with long scrolling lists, often quickly causing information overload as the size of the system increases. Furthermore, instead of providing always-available call hierarchy views these tools require the user to trigger the view for each call graph step they wish to explore, introducing a significant activation effort which results in many developers avoiding structural tools.

Some recent work in structural navigation tool support has addressed these issues indirectly. For instance, tools that record (or allow the developer to record) the navigation history reduce information overload by retaining only the most relevant program elements [24], [25], [26], [10]. While these tools offer a great reduction in information overload after relevant elements are found developers still suffer from overload during each exploration step, as the list of related elements (e.g., callees) is not filtered.

The Code Bubbles [27] approach for Eclipse and its Visual Studio counterpart, Debugger Canvas, provide a structurally oriented replacement for the standard editor by displaying relevant methods as connected boxes. The user study for Developer Canvas by DeLine et al. [28] reported both positive and negative feedback from professional developers. The authors concluded that this representation would be more beneficial as an add-on rather than a replacement for the existing IDE functionality. This matched our own conclusion that an enhancement displaying structural relationships was the best way to improve structural navigation in Visual Studio.

More relevant are the recently created navigation tools Stacksplorer [7] and Blaze [29], two always-available call graph navigation tools that have been shown to decrease maintenance task times [30]. Like *Prodet*, these tools reduce activation effort by always displaying the call graph surrounding the currently selected method. Unlike *Prodet*, these views do not filter the call graph, thus failing to reduce information overload. Additionally, these views implement a depth-only and breadth-only approach to visualizing the call graph, respectively, which can lead to backtracking during navigation. *Prodet* adopts the always-on approach pioneered by Stacksplorer and Blaze, yet it also employs filtering to reduce overload and adopts a hybrid approach to visualization that avoids backtracking.

While no structural navigation tool currently ranks or filters its output, there has been significant work on recommendation systems that recommend the next, most-relevant element to explore [31], [11]. We have adapted this work, including recommendations based on call graph topology [32] and on the textual similarity between elements [33], [17], to rank and filter structural views. Our work extends this research by including live context, or all recent developer activity including IDE events and visited source code elements.

B. Developer Practice Studies

Studies of programmer effectiveness observed differences between developers performing the same task. A study by Robillard et al. found several key behaviors effective programmers use [1]. They observed that "successful subjects performed mostly structurally guided searches (e.g., keyword and cross-reference searches), rather than browsing or scrolling." This finding inspired our work, as well as the authors of Stacksplorer and Blaze, to encourage developers to prefer structural navigation over ad hoc navigation.

Other studies found that more effective programmers use mental models of code structure. Litman et al. found that programmers successful in implementing a bug fix created a mental model of the code by studying the code systematically and comprehensively before designing the implementation [4]. LaToza et al. identified problems programmers face who maintain software [5]. These problems include developing and communicating the programmer or team's mental model of the otherwise undocumented code architecture to new team members.

Studies of developer behavior highlighted several opportunities for developers to become more effective by improving their navigation practices. Ko et al.'s study [34] of programmers performing software maintenance tasks indicated that a significant component of developers' time was spent in post-query navigating and many of the navigation steps were repeated. Revisiting methods was also highlighted by Singer [18] as a common occurrence. Similar to Robillard et al.'s finding, Fritz et al. observed in a video-based study that developers take less time to complete a task if their navigation is more structured [35].

Other studies highlight scrolling through adjacent methods as another common developer behavior in [11],[34], and [36]. These observed behaviors conflict with results from the previously discussed effectiveness studies, highlighting a gap between common developer tendencies towards unstructured exploration and the structural navigation practices of more effective developers.

V. Conclusion and Future Work

When performing software maintenance, studies suggest that a structural approach to investigating source code is more effective than an opportunistic one. Nevertheless, many developers still rely on non-structural navigation methods. To encourage the use of structural navigation, we created a Visual Studio plugin, called *Prodet*, that presents an always-on, navigable visualization of the relevant call graph neighborhood surrounding the code at hand. The tool uses a novel call graph visualization scheme that presents a balance between the depth-first and breadth-first strategies used by prior tools. This balance is enabled through the use of various recommendation algorithms to display only those methods relevant to the user's task, thus reducing their information overload.

To evaluate our approach, we conducted a longitudinal field study where we instrumented developers' working environments to collect structural navigation command and *Prodet* usage data. The data were collected for three weeks before and after *Prodet* deployment. The results show that the installation of *Prodet* had a significant effect on the use of structural

navigation, increasing it by more than two times on average. Furthermore, this effect was not limited to *Prodet* interactions, as developers using *Prodet* increased their usage of built-in navigation commands as well. Evaluating the continuous use of *Prodet* in a real-world setting with professional developers shows that the application of recent research advances can have a measurable impact on practice.

We envision two main avenues for future work: improving the visualization, and further analysis of the collected user data. To improve the visualization we plan to integrate it more tightly with the code editor. This integration should reduce the friction of referring to a separate view, but it presents significant layout challenges. To further explore our detailed usage dataset we will focus on identifying and analyzing how navigation behaviors vary in higher-level activity patterns, such as bug fixing or new feature implementation.

VI. Acknowledgments

The authors would like to thank the participants in the study and ABB Corporate Research for supporting this work.

References

[1] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Trans. on Softw. Eng.*, vol. 30, no. 12, pp. 889–903, Dec. 2004.

[2] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proc. of FSE*, 2006, pp. 23–34.

[3] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance," *IEEE Trans. on Softw. Eng.*, vol. 34, no. 3, pp. 407–432, May 2008.

[4] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Jour. of Syst. and Softw.*, vol. 7, no. 4, pp. 341–355, 1987.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proc. of ICSE*, 2006, pp. 492–501.

[6] J. Kurz, "Blaze: Navigating source code via call stack contexts," Bachelor's Thesis, 2011.

[7] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: call graph navigation helps increasing code maintenance efficiency," in *Proc. of UIST*, 2011, pp. 217–224.

[8] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: An extensible local code search framework," in *Proc. of FSE*, 2012, pp. 15:1–15:2.

[9] J. I. Maletic, M. L. Collard, and A. Marcus, "Source code files as structured documents," in *Proc. of IWPC*, 2002, pp. 289–292.

[10] M. Kersten and G. C. Murphy, "Mylar: A degree-of-interest model for IDEs," in *Proc. of AOSD*, 2005, pp. 159–168.

[11] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. M. Burnett, and R. K. E. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models." in *Proc. of VL/HCC*, 2011, pp. 109–116.

[12] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *Proc. of FSE*, 2005, pp. 11–20.

[13] X. Shen, B. Tan, and C. Zhai, "Context-sensitive information retrieval using implicit feedback," in *Proc. of SIGIR*, 2005, pp. 43–50.

[14] P. Mylonas, D. Vallet, P. Castells, M. FernAndez, and Y. Avrithis, "Personalized information retrieval based on context and ontological knowledge," *Knowledge Engineering*, vol. 23, no. 1, pp. 73–100, Mar 2008.

[15] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. on Softw. Eng. and Meth.*, vol. 17, no. 4, pp. 18:1–18:36, 2008.

[16] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, 1966.

[17] M. P. Robillard and T. Ratchford, "Context-sensitive ranking of dependencies for software navigation." School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2009.8, March 2009.

[18] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: Supporting navigation in software maintenance," in *Proc. of ICSM*, 2005, pp. 325–334.

[19] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proc. of WCRE*, 2004, pp. 214–223.

[20] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proc. of AOSD*, 2007, pp. 212–224.

[21] X. Ge, D. Shepherd, K. Damevski, and E. Murphy-Hill, "How developers use multi-recommendation system in local code search," in *Proc. of VL/HCC*, 2014.

[22] W. Snipes, A. R. Nair, and E. Murphy-Hill, "Experiences gamifying developer adoption of practices and tools," in *Proc. of ICSE SEIP*, 2014, pp. 105–114.

[23] P. Rao, *Statistical Research Methods in the Life Sciences*. Duxbury Press, 1997.

[24] D. Janzen and K. D. Volder, "Navigating and querying code without getting lost," in *Proc. of AOSD*, 2003, pp. 178–187.

[25] V. Sinha, D. Karger, and R. Miller, "Relo: Helping users manage context during interactive exploratory visualization of large codebases," in *Proc. of VL/HCC*, 2006.

[26] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *Proc. of ICSE*, 2002, pp. 406–416.

[27] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 455–464. [Online]. Available: http://dx.doi.org/10.1145/1806799.1806866

[28] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger canvas: Industrial experience with the code bubbles paradigm," in *Proc. of ICSE*, 2012, pp. 1064–1073.

[29] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers, "Blaze," in *Proc. of ICSE.*, 2012, pp. 1457–1458.

[30] J.-P. Krämer, T. Karrer, J. Kurz, M. Wittenhagen, and J. Borchers, "How tools in ides shape developers' navigation behavior," in *Proc. of CHI*, 2013, pp. 3073–3082.

[31] C. Parnin and C. Gorg, "Building usage contexts during program comprehension," in *Proc. of ICPC*, 2006, pp. 13–22.

[32] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *Proc. of ICSE*, 2007, pp. 780–783.

[33] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *Proc. of ASE*, 2007, pp. 14–23.

[34] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. on Softw. Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[35] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proc. of FSE*, 2014, pp. 7–18.

[36] M. Robillard and G. Murphy, "Automatically inferring concern code from program investigation activities," in *Proc. of ASE*, October 2003, pp. 225–234.