2014 IEEE International Conference on Software Maintenance and Evolution

CoMoGen: An Approach to Locate Relevant Task Context by Combining Search and Navigation

Katja Kevic, Thomas Fritz
University of Zurich
Department of Informatics
Zurich, Switzerland
Email: {kevic, fritz}@ifi.uzh.ch

David C. Shepherd
Industrial Software Systems
ABB Corporate Research
Raleigh NC, USA
Email: david.shepherd@us.abb.com

***Abstract*—Developers spend a substantial amount of time searching and navigating source code to locate the relevant places for performing a change task. While the searching and navigating are highly intertwined and related, most current approaches focus either on search or on navigation support for developers, keeping the two distinct. In this paper, we present an approach called CoMoGen that combines search and navigation by expanding, ranking and visualizing search results with navigation context. In an experimental analysis we found that our approach is able to generate small task-relevant context models that locates more relevant search results than state-of-the-art and state-of-the-practice search approaches. A small, preliminary user study with ten participants further yields promising preliminary findings that CoMoGen supports developers in better understanding and assessing the relevance of search results and in reducing navigation steps.**

I. INTRODUCTION

When performing a change task, developers have to locate the places in the code that have to be altered. Since it is generally infeasible to understand the whole system, developers usually follow an as-needed strategy, trying to find and understand only the parts of the system relevant to perform the change task [27]. To locate these relevant places, developers continuously search and navigate the code base, identifying potentially relevant elements using a search tool and then following structural relations in the source code to explore the context of a result and determine its relevancy [24], [42]. In this process, developers built mostly implicit mental models of task-relevant code elements and relations—more generally known as task contexts—that help the developers to make the code changes and complete the task [26], [24], [32].

There has been a substantial amount of research to support developers in identifying and capturing places in the code relevant to the task at hand (Section II). This research either focuses on supporting developers in the search for relevant starting points in the code, also known as feature location (e.g., [13], [15], [29]), or it looks into aiding developers in their navigation by recommending relevant elements once a developer has identified a starting point (e.g., [41], [36]). While developers continuously switch between search and navigation to build the relevant task context, research predominantly treats search and navigation as distinct activities. This distinction requires the developer to continuously switch between windows and tools and mentally keep track of the relevant code elements and relations.

In this paper, we present an approach called CoMoGen that combines search and navigation to support developers in the creation of initial task context models—relevant code elements and relations (Section III). CoMoGen uses initial search results as seeds, expands these by exploring their structural relations, automatically groups and ranks the resulting structural graphs and presents them to the developer. Different from current approaches with explicit task context that create the context after a developer navigated it (e.g., [43], [23]), our approach creates context models when a developer starts working on a task.

Our hypothesis is that by combining the search results with structural navigation context and presenting them in diagrams, we are able to provide more relevant search results and eliminate navigation steps developers would otherwise perform. To evaluate this hypothesis, we performed an experiment comparing our approach to state-of-the-art and state-of-the-practice approaches and a small, preliminary user study with ten participants (Section IV). The results support our hypothesis and show that CoMoGen is able to generate relevant initial context models for change tasks which contain more relevant code elements than state-of-the-practice and state-of-the-art search approaches. The preliminary user study also provides early evidence that CoMoGen helps developers in better recognizing relevant code elements and that it can significantly reduce the navigation performed per task.

This paper makes the following research contributions:

- It introduces CoMoGen, an approach that combines search with navigation to automatically generate initial context models.
- It presents a prototype and an evaluation of the approach, providing evidence that CoMoGen is effective at generating context relevant for change tasks and supports developers in their initial investigation of change tasks.

II. RELATED WORK

A substantial amount of research addresses the difficulty of finding, understanding and capturing source code to perform a change task. Work related to ours can be categorized into three areas: approaches that aid developers in searching the code base for points of interest (Section II-A), approaches that support code navigation (Section II-B) and, finally, approaches

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.28

IEEE computer society

that capture task context more explicitly (Section II-C). Since the field of related work is immense, in particular with respect to search support, we do not claim to provide an exhaustive list of related research, but rather discuss some of the most influential and most related work in the field.

A. Searching the Code Base

There is a variety of software search tools, also known as feature location tools, such as Google Eclipse Search [38], SNIAFL [52], or Sourcerer [9] that can provide a starting point for a developer's navigation. A good overview is presented by Dit and colleagues [13]. Generally, research on supporting code search can be divided by the information and input that is used to leverage the search into dynamic, static and hybrid approaches. Dynamic approaches require, for instance, test cases or a click on a system's user interface as search input, and use execution traces of a program to suggest potentially relevant elements in the code (e.g., [49], [16], [44]). More closely related to our research are code search approaches that mine static information, such as change tasks, source code and version history and that mainly differ in the applied preprocessing activities, the selection of indexing units, the similarity definitions and the granularity of the results (e.g., [52], [7], [29], [28], [9], [38], [46]). Many of these approaches require a lexical query input and use information retrieval (IR) techniques to identify and recommend relevant elements for a change task at hand. Finally, hybrid approaches combine dynamic and static approaches (e.g., [15], [8]).

All of these approaches typically present the results as an ordered list without providing any further contextual information to assess the relevance of the results without leaving the interface of the search approach. Few tools help the developer to assess the relevancy of search results by providing some additional information, such as descriptive concept labels to categorize the results [37], facets of structural information to filter the results [48], or a structural rationale for each result [21]. An exploratory study by Starke and colleagues has shown that search tools generally provide too many results, many of them irrelevant and that more contextual information would be valuable to help developers judge the relevance of the results [47].

Portfolio by McMillan and colleagues [30] is a search approach most closely related to our work that already provides some very basic call graph visualization along with the result list. Similar to our approach, Portfolio uses an IR approach to identify task-relevant code elements. In a second step, it applies PageRank and spreading activation network algorithms to identify elements related to the initial search and finally, displays all results in a flat ranked list and in one big call graph. Our approach differs in that it focuses on generating and visualizing relevant initial task context models that support developers in understanding and assessing the relevance of the results and minimizes navigation steps. CoMoGen uses a novel query expansion and result synthesis algorithm to integrate the navigation into the search process and it presents each result as a structural graph that captures more contextual information and information on various levels of granularity.

B. Code Navigation

Once an element worthy for further exploration has been identified, developers usually use navigation tools to follow structural dependencies and understand its structure, its behavior and its effects on other elements [42], [24]. Developers, which are unfamiliar with the system, have to rely on their intuition and luck to select relevant ones within the many dependencies [41]. Therefore, several approaches have been suggested to proactively recommend relevant next steps to navigate to. These approaches mainly differ in the information used to infer the recommendations from, including the use of program structure topology [41], [25], the mining of change histories [51], [53] and the use of information retrieval on the programmer's activities within the source code [50]. Further approaches combine several factors including structure and lexical information [36], [19], or even leverage recency information to suggest elements to visit next [34]. Many of these approaches only recommend a single step outwards. However, these approaches could be used complementary with our approach and we plan to investigate an integration of these navigation recommendation techniques into our approach in future work. Our approach combines the navigation with the search by using search strings to filter navigational context, as well as using control flow information to rank search results and provide more context for each search result.

C. Explicit Task Context

During change tasks developers work with a set of task-relevant code elements–also known as task context—which can become difficult to keep track of. Several approaches have been proposed to support developers in explicitly capturing these task contexts ranging from manually identifying the relevant code elements [43], [11], [35], to automatically inferring task context from a developer's interaction with the integrated development environment (IDE) [23], [22], [12], [10]. JRipples, for instance, is an approach that supports a developer in the bookkeeping and recommendation of relevant elements, after a developer identified at least one relevant starting point [11]. On the other end of the spectrum, Mylyn automatically creates a task context from a user's interaction history, ranks the elements by taking into account the frequency and recency of interactions and then uses the ranked elements to focus the UI on the task at hand [23]. The automatically captured task context has also been used to then recommend where to navigate next in the code [12].

All of these approaches capture task-relevant elements after a developer has identified or navigated them. Our approach is different in that it attempts to identify relevant task context before a developer has identified and navigated the code.

III. Approach

To support developers in their search and navigation for change tasks, we developed an approach that combines the search and the navigation and automatically generates structurally connected task context models. Our approach, CoMoGen, takes as input a user query and outputs a ranked list of connected call trees. The approach is composed of three phases: search, navigation, and visualization. An overview of the three phases is shown in Figure 1.

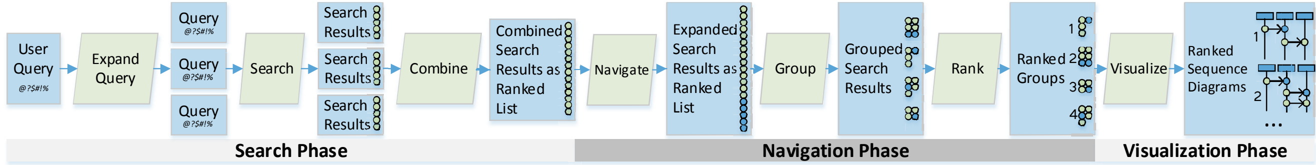
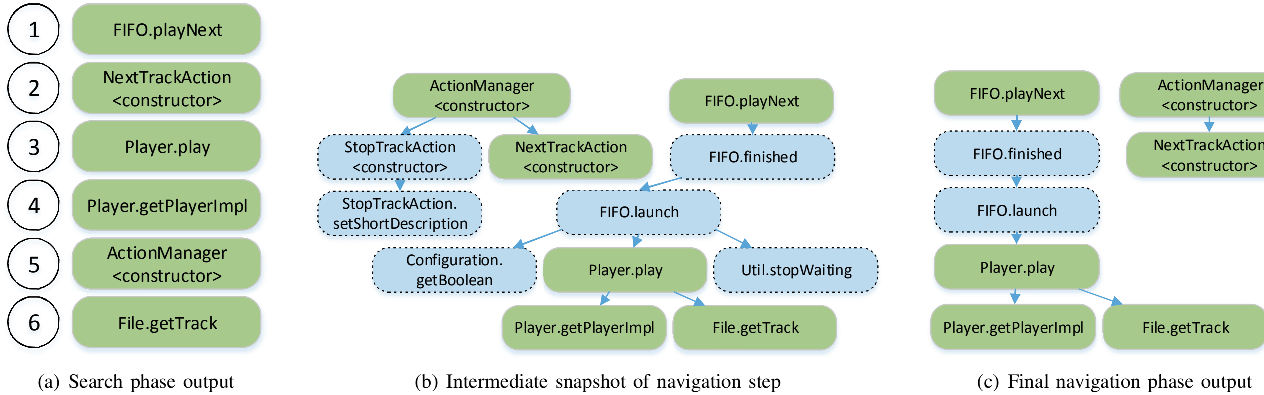


Fig. 1. An overview of the CoMoGen approach and its three phases of search, navigation and visualization.



(a) Search phase output (b) Intermediate snapshot of navigation step (c) Final navigation phase output

Fig. 2. CoMoGen output at different stages of the search and navigation process (green methods are seeds, blue ones are non-seeds).

A. Search Phase

The search is split into three steps, the expansion, the execution of the search and the synthesis of the search results. The expansion step takes an initial query and creates a set of queries that contains all possible subsets of terms from the original query, transforming a query such as “play next” into the set {“play”, “next”, “play next”}. This idea is based on the observation by Ko *et al.* [24] that a lot of user searches fail to return anything of interest. The expansion step includes the systematical combination of words from the initial query.

The next step takes the created set of queries and executes them using a state-of-the-art code search technology [46], outputting lists of relevant methods. For this code search step and prior to the actual search, the code base is indexed by extracting identifiers from each method also splitting compound identifiers into single identifiers—a common step during software search [17]—and using these identifiers to index the method. When a code search is performed, the popular Vector Space Model is employed to determine methods that are most similar to the query. Specifically, the term frequency-inverse document frequency (tf-idf) is used to determine the importance of indexes. Finally, the most similar methods together with their search score are returned as the search result. For instance, a query for “play next” locates methods such as `FIFO.playNext`. This technique is used because researchers have observed that relevant code will often contain identifiers that hint at its purpose [29] and that users will leverage these information *scents* to guide their search [36].

The final step in the search phase collects all search result lists, one for each query, and combines them together into a single ranked list. The “play next” result list may contain `FIFO.playNext` while the “next” result list may begin with `FIFO.next` but also contain `FIFO.playNext`. To fairly combine these possibly overlapping ranked result lists we use Team-Draft Interleaving [39]. This approach operates as a draft, choosing the highest remaining item from each result list during each round, resulting in a list similar to Figure 2(a). While our query expansion and result synthesis are novel contributions, search execution is simply an invocation of a state-of-the-art code search technique.

B. Navigation Phase

The navigation phase is split into three steps, the navigation outwards from seeds to discover new relevant methods, the grouping of all found methods into call trees, and the ranking of these call trees. The navigation step takes search results as seeds and explores all call trees rooted for each seed to a depth of 5, so it explores five call edges outwards from every search result. If a method discovered in this exploration is not already in the list of relevant methods, it is added as a non-seed method. For instance, in our simple example, starting at the seed `FIFO.playNext`, shown as a green node in Figure 2(b), CoMoGen explores elements related through method calls and finds the blue non-seed methods `FIFO.finished`, `FIFO.launch`, `Configuration.getBoolean` and `Util.stopWaiting` that are added to the expanded list.

The grouping step takes the expanded list of methods and groups them into connected call trees. The previously unconnected search results in Figure 2(a) plus the methods discovered through navigation are transformed into Figure 2(c). Based on the observation that developers are more effective when following structural paths [42], results are grouped into connected trees that now contain seeds along with structurally related methods. To group results into call trees, CoMoGen chooses the first method from the original search result list generated in the search phase, creates a new tree and inserts

it, and then adds related methods from the expanded list to this tree until no more methods can be added, repeating this process until all methods in the original result list are grouped. At the end of this step, CoMoGen prunes all branches that do not add any seed methods, i.e. methods that were not part of the initial search results and thus have no search score associated. As an example, the branch with the blue nodes `StopTrackAction <constructor>` and `StopTrackAction.setShortDescription` in Figure 2(b) is pruned from the navigation phase output in Figure 2(c), since it adds no “relevance” in the form of additional green nodes to the tree, while the methods`FIFO.finished` and `FIFO.launch` are kept since they build a path from `FIFO.playNext` to another seed `Player.play`.

The ranking step transforms the set of connected trees into a list ranked by relevance. CoMoGen calculates a relevance score for each tree by summing the search scores of each method element in the tree. Note that all method elements found in the initial search phase have an associated search score whereas elements found only in the expand step will not have an associated search score, as they did not match against any search queries. In our example in Figure 2(c), the group starting with `FIFO.playNext` is ranked highest because the sum of its methods’ relevance scores is highest. This ranking scheme favors highly structurally connected trees, which again reflects the fact that effective developers tend to follow structural relations and spend less time for completing a change task when doing so [42]. Note that the relevance score for a given method is its relevance with respect to its respective query from the search step.

C. Visualization Phase

The visualization phase uses ranked call trees as input and presents them as diagrams to the developer. In particular, each call tree is presented as one ‘summarized’ sequence diagram that includes objects, life lines and method calls, but omits details such as alternative combination fragments and activation blocks so that developers may quickly scan and uptake the presented information. An example of summarized sequence diagrams visualized in our prototype is presented in Figure 3. We chose UML-like diagrams, since a study by Dzidek and colleagues showed that developers are more effective when given access to UML diagrams [14]. Also, these summarized sequence diagrams contain several levels of granularity, in particular class names and method calls, to support developers in quickly grasping the presented results.

The sequence diagrams are created by traversing the ranked call tree, starting at the root of a given connected tree and performing a depth first traversal. For each class visited for the first time, a new lifeline is created and for each newly visited method call, a new message is created. In our visualization, we only have one lifeline per type to limit the size of the presented diagrams, allowing for easier uptake by the developer and to decrease response time of our interactive approach by avoiding a deep analysis of code to differentiate between different instances of a type.

D. CoMoGen Prototype

We have developed a prototype of CoMoGen—COntext MOdel GENerator—as a Visual Studio extension. Figure 3 depicts a screen shot of the prototype with the top results after running the query with the term “log” on the Sando project code base [1]. After running the query by pressing “Enter” or the Search button in the top right corner, CoMoGen executes the search, navigation and visualization phase described above and visualizes the found connected subgraphs as summarized sequence diagram ranked by their relevance to the user. Next to the visualization of the automatically generated context models that contain information on the relevant types and method calls, the prototype also contains information on the root element of the call tree, method `LogEvent.TestLogging` for the first context model in the screen shot in Figure 3. Furthermore, the prototype marks each recommendation with a bar that indicates the estimated relevancy of the recommendation (the green bar in the upper left corner, left of the root element), and a short textual summary of the diagram. Finally, the prototype provides visual adjustment options, such as zooming, direct linking of the presented elements in the code by clicking on an element in the view, and a “Generate Diagram” button to generate a full sequence diagram on demand using Microsoft’s UML SDK. A video of the prototype used in our study can be found at [6].

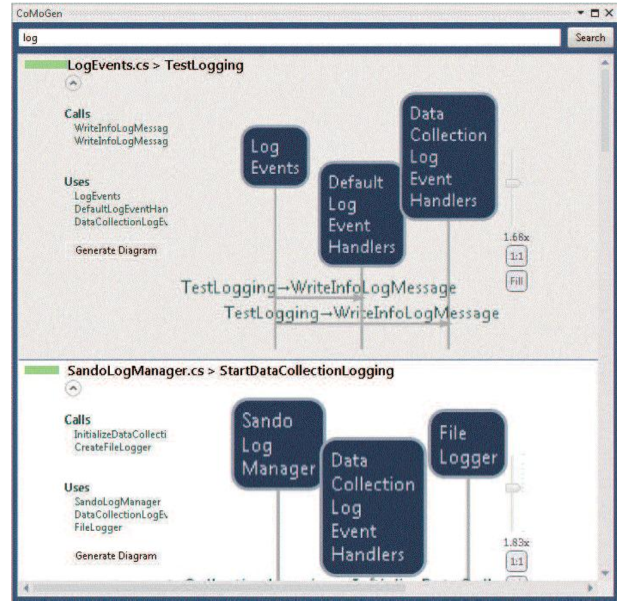


Fig. 3. Screen shot of the CoMoGen prototype.

The current implementation of CoMoGen executes most queries in only a few seconds on code bases of 18 to 50K LOC. Both the indexing and the call graph construction require a preprocessing of the project, which can take from 1 up to 3 minutes. Fortunately, this preprocessing can be completely eliminated if the indexing and call graph construction are done incrementally, which we are currently implementing for our prototype.

IV. Evaluation

To evaluate if CoMoGen supports developers in their search and navigation and the creation of initial task context models, we investigate the following three research questions:

RQ1 Is CoMoGen able to generate relevant initial task context for developers performing a change task?
RQ2 How well does CoMoGen locate initial task context in terms of code elements relevant for performing change tasks compared to state-of-the-art and state-of-the-practice search approaches?
RQ3 How well does CoMoGen support developers in the process of identifying a starting point for a change task compared to the state-of-the-art?

We performed an experimental analysis to investigate the first and second research question and a small, preliminary user study for the third research question. Note that we explicitly did not choose to perform a full user study since usability issues of beta quality software tools confound evaluations as we will point out in our preliminary user study. We expect a beta quality tool to be hardened for at least a year before a realistic user study is valid (e.g., Mylyn's user study [23]). Nonetheless, we recognize the value that user studies will add in terms of confirming the benefits of this approach on change tasks, and therefore plan to conduct them as a next step.

A. Experimental Analysis

For the experimental analysis, we used and analyzed data that we gathered in an exploratory study with ten developers that successfully performed change tasks on open source systems. More details on the exploratory study can be found elsewhere [2], [18].

Method
In the exploratory study we had a total of three open source systems, FreeMind [3] (52.5k NCLOC), Java PasswordSafe (JPass) [4] (13.5k NCLOC) and Rachota [5] (18k NCLOC), and one change task per system. Each of the ten study participants was randomly assigned one of the three change tasks. All study participants had a background in software engineering: five participants worked in a company, four were graduate students and one was a faculty member in Computer Science. The participants had an average of 11.4 years of programming experience, and on average 4.9 years of professional programming experience. Several techniques were used to capture the participants' behavior, such as participant observation, screen recording and access to the actual workspace of the participants after performing the change task. For each change task and participant we gathered a *task context*—the classes and methods the developer navigated or changed while working on the change task—, the set of classes and methods actually changed for completing the task and the search strings used while working. For each change task, we also determined one search string that was used most prevalently by the developers that completed the particular change task.

To examine research question RQ1, we compared the context models generated by CoMoGen with participants' task contexts. In particular, for each change task, we ran a search in our CoMoGen prototype using the search string determined as most prevalent, transcribed the generated context models and compared them with the task contexts gathered from the participants' navigation and changes in the IDE. To analyze whether CoMoGen generates relevant initial context models, we focus on the precision of the context models and their overlap with participants' task contexts.

To answer research question RQ2, we again used the most prevalent search string for each project and compared the search results generated by CoMoGen to the ones returned by Sando [1], [46], to represent the state-of-the-art (SoA), and the ones by Visual Studio's built-in regular expression search, to represent the state-of-the-practice (SoP). When choosing a SoA tool to compare against, we surveyed feature location tools that also leverage static information. Unfortunately, the few existing approaches were either not available, which is a well-known problem when evaluating against feature location tools [13], not search-based [19], or not effective without further refinement [45]. In our effort to compare against an effective available solution we chose Sando, a code search approach for Visual Studio. Sando implements a vector space model approach that performs as well as other leading IR approaches [33] and employs most known incremental improvements, such as favoring certain types of identifiers. While the SoA and SoP tools are not primarily focused on providing an initial starting context for a change task, we perform this comparison to gather insights on the relevance of CoMoGen's results. To compare the various approaches, we chose to calculate precision, recall and the F-measure—the harmonic mean of precision and recall. Precision represents the fraction of relevant elements found by an approach divided by the total number of elements found. Recall represents the fraction of relevant elements found divided by all possible relevant elements. The F-measure balances the competing needs of precision and recall. Furthermore, F-measure is one of the most used measures for feature location and code search evaluations [13], [20], [31] and facilitates cross-study comparisons. To fairly compare the results of the different approaches, we decided to limit the number of results we considered for each approach, as is typically done during search evaluations [20], and only consider the top 10 results from Sando and the top 5 sequence diagrams from CoMoGen. Since the SoP tool returns unranked results we had to consider all results. Note, however, that the limit had no practical impact, since no result list was ever shortened.

In this experimental analysis, we chose to evaluate the results generated by the techniques against each developer's task context to investigate whether we can provide relevant initial context models to each developer. We did not evaluate the results against an oracle determined by a set of experts as studies have shown that even experts have low agreement (about 34%) when tagging a golden set for a task [40]. The task contexts gathered in our study show a very similar low agreement rate per task with an average of 35.2% overlap on class level and 27.5% on method level. Finally, we did not pick a "best" task context—the task context of the developer performing best—since we made sure that all of our participants had sufficient programming experience and our results show that different developers solve tasks differently even if they perform equally well.

Results
RQ1. To investigate whether CoMoGen is able to generate relevant task context for developers performing a change task, we compared the code elements in the context models generated by CoMoGen using the most prevalent search string, with the code elements in each task context gathered from a participant working on the task. Table I presents the results over all context models, the first and the best context model

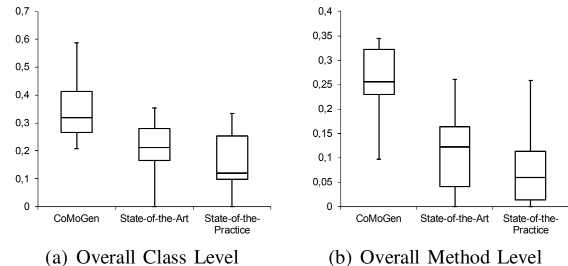
		avg. size of participants' task context	number of all CoMoGen suggestions	avg. size of CoMoGen model	over all context models			first model			best model			
					Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure	Pos.
CLASS	FreeMind	23.5	5	1.7	0.70	0.17	0.27	0.25	0.02	0.04	1.00	0.09	0.17	2
	JPass	14	8	1.6	0.83	0.64	0.72	0.67	0.12	0.20	1.00	0.20	0.34	4
	Rachota	14.8	5	2.5	0.60	0.23	0.33	0.50	0.15	0.24	1.00	0.07	0.13	2
METHOD	FreeMind	35.3	9	3	0.69	0.18	0.29	0.33	0.03	0.05	0.81	0.09	0.17	2
	JPass	24.8	14	2.8	0.31	0.21	0.25	0	0	0	0.63	0.08	0.14	5
	Rachota	41.5	20	10	0.28	0.24	0.26	0.29	0.19	0.23	0.29	0.19	0.23	1

TABLE I. PRECISION AND RECALL OF CONTEXT MODELS GENERATED BY COMOGEN WITH RESPECT TO PARTICIPANTS' TASK CONTEXT

generated per query. On average, the ten study participants had a mean of 17.2 classes and 33.9 methods in their task context. In comparison, CoMoGen generated 10 context models over all three change tasks, an average of 3.3 context models per query, each with an average of 1.8 classes and 4.3 methods, almost ten times smaller than the actual task contexts. In total, we examined 33 combinations of context models and task context. Out of these combinations, 93.9% of the cases had at least one overlap between the generated context model and the inspected task context on class level, and 75.8% on method level. This shows that most of the generated context models contained relevant code elements that were navigated by the participants performing the change task. A precision ranging from 0.6 to 0.83 on class level and 0.28 to 0.69 on method level, further shows that much of the presented context models was relevant to the participant, in particular on class level. Considering only the best context model per change task, the one that contained the most relevant code elements, precision on class level goes up to 1.0 for all three projects and also increases on method level. For the first result presented by CoMoGen, the precision drops, but is still reasonable. Since our approach attempts to generate initial context models, it is important to capture some relevant parts and not presenting too many irrelevant code elements. Therefore, a high precision is preferred over a high recall, especially given the big differences and low overlap between different developers' contexts. The high overlap between the generated context models and developers' contexts and the reasonably high recall, especially on class level, indicate that CoMoGen produces small, but quite precise context models and illustrate the high potential of our approach.

RQ2. To investigate whether CoMoGen can locate relevant additional code elements with respect to SoA and SoP search approaches, we compare the F-measures of CoMoGen to the ones of Sando and to the ones of Visual Studio's built-in regular expression search tool (VSS). Since Sando (SoA) and VSS (SoP) focus solely on the search and might thus return fewer results than CoMoGen which attempts to generate initial context models, also adding relevant navigation context, we calculated the F-measure to balance out recall and precision.

Figure 4 presents the F-measures on class and method level for the search results produced by all three approaches. Over all three change tasks and both levels—class and method level—CoMoGen performs better than the SoA and the SoP approach. On class level, CoMoGen has a median F-measure of 0.32 (IQR_{CMG} = 0.15) compared to Sando's median of 0.21 (IQR_{SoA} = 0.11) and Mdn_{SoP} = 0.12 (IQR = 0.16) for VSS. On method level, CoMoGen has a median of 0.26 (IQR_{CMG} = 9.2), compared to a median of 0.12 (IQR_{SoA} = 0.12) for Sando and Mdn_{SoP} = 0.06 (IQR_{SoP} = 0.10) for VSS. A Wilcoxon signed-rank test, a non-parametric paired samples test, shows that CoMoGen performs significantly better than Sando on class (p = .028) and method level (p = .005), and it also performs significantly better than VSS on class (p = .007) and method level (p = .013). Just looking at the recall, CoMoGen performs significantly better than SoA and SoP over all three projects on class (p = .018 for SoA and p = .007 for SoP) and method level (p = .005 for SoA and p = .013 for SoP). Thus, at both, class and method level, CoMoGen added relevant code elements to the search results compared to a state-of-the-art and a state-of-the-practice approach.



(a) Overall Class Level (b) Overall Method Level

Fig. 4. F-measures of results generated by CoMoGen, the SoA and the SoP approach on class and method level.

As previously mentioned, we calculate F-measures for participants' task contexts since the intent of CoMoGen is to support developers during the search and the navigation phase. However, for completeness we also calculated the F-measures for the retrieved search results with respect to the actual changes made by the participants for each change task in the exploratory study. The actual changes usually contain fewer code elements than the task contexts do, which comprise all elements navigated to, not just changed ones. Again, CoMoGen recommended significantly more relevant results on class level (Mdn = $0.35, IQR$ = 0.18) than SoP (Mdn = $0.12, IQR$ = 0.23) with p = .007. Also, the median of the collected F-measures for CoMoGen's search results is higher than the median of the SoA tool (Mdn = $0.26, IQR$ = 0.29), but this difference is not significant p = .138. On method level, CoMoGen recommends significantly more relevant elements (Mdn = $0.26, IQR$ = 0.24) than both, the SoP approach (Mdn = $0.08, IQR$ = 0.11) with p = .012, and the SoA approach (Mdn = $0.11, IQR$ = 0.22) with p = .028.

B. Small and Preliminary User Study

To investigate if our approach provides an actual benefit to users in the initial phase of performing a change task, we conducted a small, preliminary user study with ten university students. In this study, we compared our approach, specifically the prototype we implemented of our approach, to the state-of-the-art search approach Sando.

Method
In the preliminary user study, we asked ten undergraduate and graduate students to investigate six change tasks on a small code base they were unfamiliar with and to identify relevant starting points in the code. All ten participants were male students with a computer science major, each had a profound background in object-oriented programming (average (Avg) of 7.4 years and standard deviation (SD) of 3.1 years) and at least some professional software development experience ($Avg = 2.9 years, SD = 2.2 years$). All participants were also familiar with the common navigation features in Visual Studio (VS), such as *Go To Definition*, *View Call Hierarchy* or *Find All References*. Each participant was given a VS instance with CoMoGen and Sando—the SoA approach for this comparison—installed, given a short introduction to Sando and CoMoGen, and allowed to try both tools for up to five minutes. Then, for each change task, the participant was asked to read the change task, use an assigned tool (either CoMoGen or Sando) to investigate the task, and identify up to three places in the source code on method level or below to which they would point a new developer who was just assigned to perform the change task. Each task was limited to ten minutes and participants were asked to think aloud. This process was repeated for three tasks before switching to the other tool and completing three more tasks. After a participant finished all six tasks, we conducted a semi-structured interview on the gathered experiences. During the tasks audio recording, screen recording, and activity capture software was utilized and written notes were taken to facilitate post-analysis.

We applied an A/B testing method with a repeated measures design and counterbalancing for possible learning effects. We assigned participants to two groups, one starting out with CoMoGen, one with Sando, switching after three tasks to then perform the last three of the overall six tasks with the other approach. The assignment was done randomly, only ensuring that participants in both groups had about equal experience in object-oriented programming, resulting in one group with a mean of 6.6 years (SD = 3.7) and the other with 8.2 years (SD = 2.5). The study was performed in our lab to ensure the same conditions for all participants.

The six change tasks were existing change tasks chosen from the Sando project, an open source project with 8.7K lines of code (revision 1350). The change tasks were the most recent existing Sando tasks that had been completed successfully and where a corresponding commit was identifiable. All six change tasks cover different parts of the functionality within Sando. For each change task, we looked up the changed methods from the commit history and considered these changed method as the most relevant starting points for a change task, which is a common procedure when evaluating feature location approaches [13]. Each change task required the editing of an average of 3 methods (SD = 1.3).

Results
In this section we present the primary findings of our preliminary user study based on an analysis of the transcription of the 8.2 hours of video and audio capture together with the data gained from the activity capture software. In particular, we examine the recognition of relevant results in the search result visualizations and the navigation and searches performed per task.

RQ3. Overall, the results indicate that CoMoGen supports developers in better recognizing relevant search results while reducing navigation steps and searches. However, there is no significant difference in the time used per task or the number of cases in which participants identified a relevant starting point when using CoMoGen versus using Sando (p = .191). In 13 out of 30 cases for CoMoGen and 15 out of 30 cases for Sando, a participant identified a relevant starting point that was in the commit of the change task.

Better recognition of relevant search results. In our study, we observed that participants were a lot more likely to miss a relevant result when they used Sando for a search than when they used CoMoGen. In the screen recordings and our notes for all 60 change task investigations, we identified 16 cases in which a relevant result was in Sando's result list and presented to the participant but not recognized as such. At the same time, there was only a single case in which a participant used CoMoGen and missed a relevant result that was presented in one of the context models to him. In the follow-up interviews, several participants further mentioned that they only went through the top most items listed in Sando's search result list when the search result list was too long. These findings suggest that CoMoGen's grouping of results and the provision of navigational context eases the recognition of relevant results and reduces the likelihood of overlooking relevant ones.

Reduced navigation. Comparing the data collected for the tasks performed with Sando with the ones with CoMoGen also shows that participants opened fewer search results and performed fewer navigation steps when using CoMoGen. A Mann-Whitney U test shows that participants using CoMoGen opened significantly fewer search results in an editor than participants using Sando ($p = .017, U = 291.0, Z = 1.307$). A median of 2 ($IQR$ = 3) search results were opened in an editor per task when using CoMoGen, compared to a median of 4 (IQR = 4) opened search results per task when using Sando. We used Mann-Whitney U tests, since the collected data is categorical, does not meet parametric assumptions, and the participants differ in each category.

We also found that participants using CoMoGen performed significantly fewer navigation steps when completing a task than with Sando, with p = .042 ($U = 312.5, Z = -2.036$) and a median of 18.5 per task (IQR = 10) for CoMoGen and 23 (IQR = 11.5) for Sando.

Since there is no significant difference in the number of cases in which participants identified a relevant starting point, our findings indicate that CoMoGen visualizes context to developers that allows to better assess the relevance of results and reduces the need to jump into the code and to navigate.

Fewer searches. Participants in our study ran significantly fewer queries when using CoMoGen than when using Sando ($p = .031, U = 305.5, Z = -2.163$). Since there is no significant

difference in the number of cases in which relevant places were identified, this finding suggests that CoMoGen might be able to generate more precise results than Sando, further supporting our findings from our experimental analysis for research question RQ2.

Similar time used per task. The time spent on each change task did not differ significantly when using CoMoGen versus Sando ($p = .642, U = 420.0, Z = -0.465$), with a median of 572 seconds ($IQR = 158.25$) for a task with CoMoGen and 600 seconds ($IQR = 178.25$) for a task with Sando. Given the significantly reduced navigation that would usually also entail a reduced time per task when using a production-quality tool, we hypothesize that this lack in a difference is due to CoMoGen's poor runtime performance and its lower usability.

C. Threats to Validity

The small size of our subject sample in our experimental analysis limits the generalizability of our results. We tried to mitigate this risk by having participants with a multitude of backgrounds and multiple years of programming experience. Further studies are needed to investigate the full potential of our approach and the generalizability of our observations.

Threats to the preliminary user study are the limited number and experience of participants, the small size of the project, the focus on Visual Studio and C#, as well as learning effects. To mitigate some of these risks, we used a repeated measures design and randomly assigned participants to groups. Also, since this was only a small, preliminary study to gather some initial evidence on the use of CoMoGen, we do not claim that these results are generalizable.

Finally, the usability difference between Sando and CoMoGen poses a threat to the validity of our user study results. For instance, the slower performance of CoMoGen for searches, might have caused participants to run fewer searches and spend more time with each result. Further usability and performance improvements and a more full-fledged study are needed to investigate the generalizability of our results.

V. Discussion

Our findings from the experimental analysis and the preliminary user study present insights into the usefulness and the usability of our approach for the automatic generation of task context models. In this section, we further discuss a set of considerations for the design and usability of our approach and the prototype we built.

A. Task Context Models

For the automatic generation of task context models, our approach has currently a strong focus on call relations and textual similarity to the query string. This strong focus, for instance, leads to the pruning of all call branches in the navigation phase that do not end in methods—seeds—with textual similarity to the query string found in the search phase. While this strong focus allows to keep the generated context models at a reasonable size, containing many relevant elements and relations, it can also result in a high number of small context models and miss other relevant information. For instance, in our experimental analysis the generated context models had an average of 4.3 method elements, compared to the navigation models of the study participants with an average size of 33.9 method elements. There are two major ways in which we intend to extend our approach in the future. First, we are planning to investigate what a good range for the size of the generated models would be for reducing further navigation steps while still being able to quickly grasp the presented information in the model. Such a range will provide valuable thresholds for the generation of context models. Second, while call relations were the most commonly followed relations in our exploratory study and the relevance of the generated context models suggests that call relations capture an important aspect, other kinds of relations and information, such as inheritance, code coupling [53] or recency [34], might provide further value for the generation of relevant context models. In the future, we intend to integrate further information and integrate research findings, especially from the program navigation domain (Section II-B).

B. A Dynamic Approach

Our current approach supports developers in determining the relevance of search results and reduces the navigation steps performed right after a search. Once a developer explores the context models further by jumping into the code, reading it and navigating outwards—something we also observed in our preliminary user study—our approach does not yet provide any support. Since approaches to explicitly capture task context have been shown to help developers in their navigation and work on a task (see Section II-C), we plan to extend our approach into a more dynamic support for the generation and capturing of task context. For instance, after the developer steps from a generated context model into the code and navigates the code, our view could automatically update the context model, add new relations and code elements based on the navigation steps taken by the developer, highlight the current position in the model, and generate and visualize automatic recommendations for further navigation based on the original query and the recent navigation. The dynamic adaptation and generation of the context model could further reduce navigation, even long after a search and at the same time provide a good overview of the task-relevant elements to the developer.

C. Tool Usability

For our preliminary user study, our prototype was fully functioning, tested and piloted beforehand. However, when we ran our study, we observed that developers lost time on several smaller bugs, usability and performance issues. In particular, we observed that the search execution in CoMoGen took longer than in Sando, switching back to the CoMoGen tab within the Visual Studio IDE caused the previously selected diagram to move out of the view until one clicked into the diagram, and scrolling was not as smooth as users expected it to be. While these usability and performance issues were still allowing the developer to fully use the tool and us to observe some benefits of our approach, we hypothesize that this negatively influenced the time used for identifying a good starting point in the study. In future work, we plan to improve the prototype's performance and usability and run a full-fledged usability study.

VI. CONCLUSION

Developers create task-relevant models while searching and navigating the code. In this work, we presented an approach (CoMoGen) that combines search with navigation to automatically generate and visualize initial context models for a task. We examined the potential of the approach and its use by developers with an experimental analysis and a small, preliminary user study. Our findings provide early evidence that CoMoGen generates small and highly relevant context models, that it outperforms state-of-the-art and state-of-the-practice search approaches, and that it supports developers in better recognizing relevant results while reducing navigation. In future work, we intend to analyze the visualization of the code context models, integrate research from the domain of navigation recommendations and develop support to also more dynamically and explicitly capture task context.

REFERENCES

[1] http://sando.codeplex.com/.

[2] https://github.com/abb-iss/study-artifacts-for-code-context-models.

[3] http://sourceforge.net/projects/freemind/.

[4] http://sourceforge.net/projects/jpwsafe/.

[5] http://sourceforge.net/projects/rachota/.

[6] http://www.screencast.com/users/davidcshepherd/folders/Jing/media/4d7e2de4-5e0f-43a1-ad3a-234a88eff3bc.

[7] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in oo systems. In *In Proceedings of the 6th Working Conference on Reverse Engineering*, pages 136–144, 1999.

[8] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 357–366, 2005.

[9] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4, 2009.

[10] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512, 2010.

[11] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. Jripples: a tool for program comprehension during incremental change. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 149–152, 2005.

[12] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the ACM Symposium on Software Visualization*, pages 183–192, 2005.

[13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[14] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. *IEEE Transactions on Software Engineering*, pages 407–432, 2008.

[15] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[16] A. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 337–346, 2005.

[17] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

[18] T. Fritz, D. C. Sheperd, K. Kevic, W. Snipes, and C. Bräunlich. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014. To appear.

[19] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 14–23, 2007.

[20] E. Hill, L. L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527, 2011.

[21] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 117–125, May 2005.

[22] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, 2005.

[23] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, 2006.

[24] 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 Transactions on Software Engineering*, 32:971–987, 2006.

[25] T. LaToza and B. Myers. Visualizing call graphs. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 117–124, 2011.

[26] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, 2006.

[27] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In *Papers Presented at the First Workshop on Empirical Studies of Programmers*, pages 80–98, 1986.

[28] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[29] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, 2004.

[30] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[31] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, 2009.

[32] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 33–48, 2005.

[33] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the IEEE 18th International Conference on Program Comprehension*, pages 68–71, 2010.

[34] C. Parnin and C. Gorg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 14–23, 2006.

[35] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the IEEE 17th International Conference on Program Comprehension*, pages 10–19, 2009.

[36] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart. Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1471–1480, 2012.

[37] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 37–48, June 2007.

[38] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with google. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 334–338, 2006.

[39] F. Radlinski and N. Craswell. Comparing the sensitivity of information retrieval metrics. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 667–674, 2010.

[40] M. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An empirical study of the concept assignment problem. Technical report, 2007.

[41] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 11–20, 2005.

[42] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, pages 889–903, 2004.

[43] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24rd International Conference on Software Engineering*, pages 406–416, 2002.

[44] T. Savage, M. Revelle, and D. Poshyvanyk. Flat3: Feature location and textual tracing tool. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 255–258, 2010.

[45] P. Shao and R. K. Smith. Feature location by IR modules and call graph. In *Proceedings of the 47th Annual Southeast Regional Conference*, pages 70:1–70:4, 2009.

[46] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: An extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 15:1–15:2, 2012.

[47] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 157–166, 2009.

[48] J. Wang, X. Peng, Z. Xing, and W. Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 762–771, Piscataway, NJ, USA, 2013. IEEE Press.

[49] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceerdings of the Conference on Software Maintenance*, pages 200–205, 1992.

[50] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, pages 159–166, 2002.

[51] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[52] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering*, pages 293–303, 2004.

[53] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.